Intel® Advanced Vector Extensions 10

Architecture Specification

July 2023

Revision 1.0

Order Number: 355989-001US

Notices & Disclaimers

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

No product or component can be absolutely secure.

Your costs and results may vary.

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

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

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

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

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

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

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

CONTENTS

PAGE

CHAPTER 1
INTEL® AVX10.1 INSTRUCTION SET REFERENCE, A-Z

FIGURES

TABLES

CHAPTER 1
INTEL® AVX10.1 INSTRUCTION SET REFERENCE, A-Z

1.1 INTRODUCTION

Intel® Advanced Vector Extensions 10 (Intel® AVX10) represents the first major new vector ISA since the introduction of Intel® Advanced Vector Extensions 512 (Intel® AVX-512) in 2013. This ISA will establish a common, converged vector instruction set across all Intel architectures, incorporating the modern vectorization aspects of Intel AVX-512. This ISA will be supported on all future processors, including Performance cores (P-cores) and Efficient cores (E-cores).

The Intel AVX10 ISA represents the latest in ISA innovations, instructions, and features moving forward. Based on the Intel AVX-512 ISA feature set and including all Intel AVX-512 instructions introduced with future Intel® Xeon® processors based on Granite Rapids microarchitecture, it will support all instruction vector lengths (128, 256, and 512), as well as scalar and opmask instructions. A “converged” version of Intel AVX10 with maximum vector lengths of 256 bits and 32-bit opmask registers will be supported across all Intel processors, while 512-bit vector registers and 64-bit opmasks will continue to be supported on some P-core processors.

The Intel AVX10 architecture introduces several features and capabilities beyond the Intel AVX2 ISA:

- Version-based instruction set enumeration.
- Intel AVX10/256 — Converged implementation support on all Intel® processors to include all the existing Intel AVX-512 capabilities such as EVEX encoding, 32 vector registers, and eight mask registers at a maximum vector length of 256 bits and maximum opmask length of 32 bits.
- Intel AVX10/512 — Support for 512-bit vector and 64-bit opmask registers on P-core processors for heavy vector compute applications that can leverage the additional vector length.
- Embedded rounding and Suppress All Exceptions (SAE) control for YMM versions of the instructions.
- VMX capability to create Intel AVX10/256 virtual machines that provide a hardware enforced Intel AVX10/256 execution environment on an Intel AVX10/512 capable processor.

1.2 FEATURE ENUMERATION

Intel AVX10 introduces a versioned approach for enumeration that is monotonically increasing, inclusive, and supporting all vector lengths. This is introduced to simplify application development by ensuring that all Intel processors support the same features and instructions at a given Intel AVX10 version number, as well as reduce the number of CPUID feature flags required to be checked by an application to determine feature support.

In this enumeration paradigm, the application developer will only need to check three fields:

1. A CPUID feature bit indicating that the Intel AVX10 ISA is supported.
2. A version number to ensure that the supported version is greater than or equal to the desired version.
3. A vector length bit indicating the maximum supported vector length.

The “AVX10 Converged Vector ISA Enable” bit will indicate processor support for the ISA and the presence of an “AVX10 Converged Vector ISA” leaf containing fields for the version number and the supported vector bit lengths. See Table 1-1 for details.

Table 1-1. CPUID Enumeration of Intel® AVX10

CPUID Bit	Description	Type
CPUID.(EAX=07H, ECX=01H):EDX[bit 19]	If 1, the Intel AVX10 Converged Vector ISA is supported.	Bit (0/1)
CPUID.(EAX=24H, ECX=00H):EAX[bits 31:0]	Reports the maximum supported sub-leaf.	Integer
CPUID.(EAX=24H, ECX=00H):EBX[bits 7:0]	Reports the Intel AVX10 Converged Vector ISA version.	Integer (≥ 1)
CPUID.(EAX=24H, ECX=00H):EBX[bits 15:8]	Reserved	N/A
CPUID.(EAX=24H, ECX=00H):EBX[bit 16]	If 1, indicates that 128-bit vector support is present.	Bit (0/1)
CPUID.(EAX=24H, ECX=00H):EBX[bit 17]	If 1, indicates that 256-bit vector support is present.	Bit (0/1)
CPUID.(EAX=24H, ECX=00H):EBX[bit 18]	If 1, indicates that 512-bit vector support is present.	Bit (0/1)
CPUID.(EAX=24H, ECX=00H):EBX[bits 31:19]	Reserved	N/A
CPUID.(EAX=24H, ECX=00H):ECX[bits 31:0]	Reserved	N/A
CPUID.(EAX=24H, ECX=00H):EDX[bits 31:0]	Reserved	N/A
CPUID.(EAX=24H, ECX=01H):EAX[bits 31:0]	Reserved for discrete feature bits.	N/A
CPUID.(EAX=24H, ECX=01H):EBX[bits 31:0]	Reserved for discrete feature bits.	N/A
CPUID.(EAX=24H, ECX=01H):ECX[bits 31:0]	Reserved for discrete feature bits.	N/A
CPUID.(EAX=24H, ECX=01H):EDX[bits 31:0]	Reserved for discrete feature bits.	N/A

The versioned approach to ISA enumeration is expected to adhere to the following rules when incrementing from version N to N+1:

- All contemporary processor families[1] support Intel AVX10 Version N+1.
- Intel AVX10 Version N+1 delivers significant value over version N to justify the associated software enabling efforts.

In the rare case of a feature needing to be introduced in-between versions, a discrete CPUID feature bit of the form “AVX10-XXXX” may be allocated and enumerated in sub-leaf 1 of CPUID leaf 24H, i.e., CPUID.(EAX=24H, ECX=01H). However, this is expected to be the exception rather than the norm due to the necessity for entrenched legacy support.

Several other important tenets regarding Intel AVX10 enumeration are as follows:

- Versions are expected to be inclusive such that version N+1 is a superset of version N. Once an instruction is introduced in Intel AVX10.x, it is expected to be carried forward in all subsequent Intel AVX10 versions, allowing a developer to check only for a version greater than or equal to the desired version.
- Any processor that enumerates support for Intel AVX10 will also enumerate support for Intel AVX and Intel AVX2.
- Developers can assume that the highest supported vector length for a processor implies that all lesser vector lengths are also supported. Scalar Intel AVX-512 instructions will be supported independent of the maximum vector width.
- For Intel AVX10/256, 32-bit opmask register lengths are supported. For Intel AVX10/512, 64-bit opmask are supported. There are currently no plans to support an Intel AVX10/128 implementation.

The initial, fully-featured version of Intel AVX10 will be enumerated as Version 2 (denoted as Intel AVX10.2). This will include the new YMM-form of embedded rounding and Suppress All Exceptions (SAE), the new enumeration scheme, and several new sets of instructions.

An early version of Intel AVX10 (Version 1, or Intel AVX10.1) that only enumerates the Intel AVX-512 instruction set at 128, 256, and 512 bits will be enabled on the Granite Rapids microarchitecture for software pre-enabling. Applications written to Intel AVX10.1 will run on any future Intel processor (P-core or E-core) that enumerates Intel

---

[1] Contemporary processor families supporting Intel AVX10 begin with future Intel Xeon processors based on Granite Rapids microarchitecture.

AVX10.1 or higher at the matching desired vector lengths. Intel AVX-512 instruction families included in Intel AVX10.1 are shown in Table 1-2.

Table 1-2. Intel® AVX-512 CPUID Feature Flags Included in Intel® AVX10

Feature Introduction	Intel® AVX-512 CPUID Feature Flags Included in Intel® AVX10
Intel® Xeon® Scalable Processor Family based on Skylake microarchitecture	AVX512F, AVX512CD, AVX512BW, AVX512DQ
Intel® Core™ processors based on Cannon Lake microarchitecture	AVX512_VBMI, AVX512_IFMA
2nd generation Intel® Xeon® Scalable Processor Family based on Cascade Lake product	AVX512_VNNI
3rd generation Intel® Xeon® Scalable Processor Family based on Cooper Lake product	AVX512_BF16
3rd generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture	AVX512_VPOPCNTDQ, AVX512_VBMI2, VAES, GFNI, VPCLMULQDQ, AVX512_BITALG
4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture	AVX512_FP16

NOTE

VAES, VPCLMULQDQ, and GFNI EVEX instructions will be supported on Intel AVX10 machines but will continue to be enumerated by their existing discrete CPUID feature flags. This requires the developer to check for both the feature and Intel AVX10, e.g., {AVX10 AND VAES}.

Intel AVX-512 will continue to be supported on P-core-only processors for the foreseeable future to support legacy applications. However, new vector ISA features will only be added to the Intel AVX10 ISA moving forward. While Intel AVX10/512 includes all Intel AVX-512 instructions, it important to note that applications compiled to Intel AVX-512 with vector length limited to 256 bits are not guaranteed to be compatible on an Intel AVX10/256 processor due to differences in the supported mask register width (see Table 1-3). Intel will develop tools to enable developers to validate their code prior to deployment.

Table 1-3. Feature Differences Between Intel® AVX-512 and Intel® AVX10

Feature	Intel® AVX-512	Intel® AVX10.1/256	Intel® AVX10.2/256	Intel® AVX10.1/512	Intel® AVX10.2/512
Maximum opmask register length	64 bits	32 bits	32 bits	64 bits	64 bits
128-bit vector (XMM) register support	Yes	Yes	Yes	Yes	Yes
256-bit vector (YMM) register support	Yes	Yes	Yes	Yes	Yes
512-bit vector (ZMM) register support	Yes	No	No	Yes	Yes
YMM embedded rounding	No	No	Yes	No	Yes
ZMM embedded rounding	Yes	No	No	Yes	Yes

1.3 INSTRUCTIONS

Instruction pages follow; all changes to existing instructions are highlighted in violet font with change bars to the left.

ADDPD—Add Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 58 /r ADDPD xmm1, xmm2/m128	A	V/V	SSE2	Add packed double precision floating-point values from xmm2/mem to xmm1 and store result in xmm1.
VEX.128.66.0F.WIG 58 /r VADDPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Add packed double precision floating-point values from xmm3/mem to xmm2 and store result in xmm1.
VEX.256.66.0F.WIG 58 /r VADDPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add packed double precision floating-point values from ymm3/mem to ymm2 and store result in ymm1.
EVEX.128.66.0F.W1 58 /r VADDPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Add packed double precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.256.66.0F.W1 58 /r VADDPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Add packed double precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.512.66.0F.W1 58 /r VADDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F OR AVX10.1[1]	Add packed double precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Adds two, four or eight packed double precision floating-point values from the first source operand to the second source operand, and stores the packed double precision floating-point result in the destination operand.

EVEX encoded versions: 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.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the first source operand is a XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. 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 ZMM register destination are unmodified.

Operation

VADDPD (EVEX Encoded Versions) When SRC2 Operand is a Vector Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC1[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
```

VADDPD (EVEX Encoded Versions) When SRC2 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] := SRC1[i+63:i] + SRC2[63:0]
                ELSE
                    DEST[i+63:i] := 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
```

VADDPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] + SRC2[63:0]
DEST[127:64] := SRC1[127:64] + SRC2[127:64]
DEST[191:128] := SRC1[191:128] + SRC2[191:128]
DEST[255:192] := SRC1[255:192] + SRC2[255:192]
DEST[MAXVL-1:256] := 0
.
```

VADDPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] + SRC2[63:0]
DEST[127:64] := SRC1[127:64] + SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

ADDPD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] + SRC[63:0]
DEST[127:64] := DEST[127:64] + SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VADDPD __m512d _mm512_add_pd (__m512d a, __m512d b);
VADDPD __m512d _mm512_mask_add_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VADDPD __m512d _mm512_maskz_add_pd (__mmask8 k, __m512d a, __m512d b);
VADDPD __m256d _mm256_mask_add_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VADDPD __m256d _mm256_maskz_add_pd (__mmask8 k, __m256d a, __m256d b);
VADDPD __m128d _mm_mask_add_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VADDPD __m128d _mm_maskz_add_pd (__mmask8 k, __m128d a, __m128d b);
VADDPD __m512d _mm512_add_round_pd (__m512d a, __m512d b, int);
VADDPD __m512d _mm512_mask_add_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VADDPD __m512d _mm512_maskz_add_round_pd (__mmask8 k, __m512d a, __m512d b, int);
ADDPD __m256d _mm256_add_pd (__m256d a, __m256d b);
ADDPD __m128d _mm_add_pd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

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

ADDPS—Add Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 58 /r ADDPS xmm1, xmm2/m128	A	V/V	SSE	Add packed single precision floating-point values from xmm2/m128 to xmm1 and store result in xmm1.
VEX.128.0F.WIG 58 /r VADDPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Add packed single precision floating-point values from xmm3/m128 to xmm2 and store result in xmm1.
VEX.256.0F.WIG 58 /r VADDPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add packed single precision floating-point values from ymm3/m256 to ymm2 and store result in ymm1.
EVEX.128.0F.W0 58 /r VADDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Add packed single precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.256.0F.W0 58 /r VADDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Add packed single precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.512.0F.W0 58 /r VADDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	C	V/V	AVX512F OR AVX10.1[1]	Add packed single precision floating-point values from zmm3/m512/m32bcst to zmm2 and store result in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Adds four, eight or sixteen packed single precision floating-point values from the first source operand with the second source operand, and stores the packed single precision floating-point result in the destination operand.

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

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the first source operand is a XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. 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 ZMM register destination are unmodified.

Operation

VADDPS (EVEX Encoded Versions) When SRC2 Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC1[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
```

VADDPS (EVEX Encoded Versions) When SRC2 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] := SRC1[i+31:i] + SRC2[31:0]
                ELSE
                    DEST[i+31:i] := 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
```

VADDPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[63:32] := SRC1[63:32] + SRC2[63:32]
DEST[95:64] := SRC1[95:64] + SRC2[95:64]
DEST[127:96] := SRC1[127:96] + SRC2[127:96]
DEST[159:128] := SRC1[159:128] + SRC2[159:128]
DEST[191:160]:= SRC1[191:160] + SRC2[191:160]
DEST[223:192] := SRC1[223:192] + SRC2[223:192]
DEST[255:224] := SRC1[255:224] + SRC2[255:224].
DEST[MAXVL-1:256] := 0
```

VADDPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[63:32] := SRC1[63:32] + SRC2[63:32]
DEST[95:64] := SRC1[95:64] + SRC2[95:64]
DEST[127:96] := SRC1[127:96] + SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

ADDPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[63:32] := SRC1[63:32] + SRC2[63:32]
DEST[95:64] := SRC1[95:64] + SRC2[95:64]
DEST[127:96] := SRC1[127:96] + SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VADDPS __m512 _mm512_add_ps (__m512 a, __m512 b);
VADDPS __m512 _mm512_mask_add_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VADDPS __m512 _mm512_maskz_add_ps (__mmask16 k, __m512 a, __m512 b);
VADDPS __m256 _mm256_mask_add_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VADDPS __m256 _mm256_maskz_add_ps (__mmask8 k, __m256 a, __m256 b);
VADDPS __m128 _mm_mask_add_ps (__m128d s, __mmask8 k, __m128 a, __m128 b);
VADDPS __m128 _mm_maskz_add_ps (__mmask8 k, __m128 a, __m128 b);
VADDPS __m512 _mm512_add_round_ps (__m512 a, __m512 b, int);
VADDPS __m512 _mm512_mask_add_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VADDPS __m512 _mm512_maskz_add_round_ps (__mmask16 k, __m512 a, __m512 b, int);
ADDPS __m256 _mm256_add_ps (__m256 a, __m256 b);
ADDPS __m128 _mm_add_ps (__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

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

ADDSD—Add Scalar Double Precision Floating-Point Values

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
F2 0F 58 /r ADDSD xmm1, xmm2/m64	A	V/V	SSE2	Add the low double precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.LIG.F2.0F.WIG 58 /r VADDSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Add the low double precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.LLIG.F2.0F.W1 58 /r VADDSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F OR AVX10.1[1]	Add the low double precision floating-point value from xmm3/m64 to xmm2 and store the result in xmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Adds the low double precision floating-point values from the second source operand and the first source operand and stores the double precision floating-point result in the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source and destination operands are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. 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 version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VADDSD is encoded with VEX.L=0. Encoding VADDSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VADDSD (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC1[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] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VADDSD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] + SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

ADDSD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] + SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VADDSD __m128d _mm_mask_add_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VADDSD __m128d _mm_maskz_add_sd (__mmask8 k, __m128d a, __m128d b);
VADDSD __m128d _mm_add_round_sd (__m128d a, __m128d b, int);
VADDSD __m128d _mm_mask_add_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VADDSD __m128d _mm_maskz_add_round_sd (__mmask8 k, __m128d a, __m128d b, int);
ADDSD __m128d _mm_add_sd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

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

ADDSS—Add Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 58 /r ADDSS xmm1, xmm2/m32	A	V/V	SSE	Add the low single precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.LIG.F3.0F.WIG 58 /r VADDSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Add the low single precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.LLIG.F3.0F.W0 58 /r VADDSS xmm1{k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F OR AVX10.1[1]	Add the low single precision floating-point value from xmm3/m32 to xmm2 and store the result in xmm1with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Adds the low single precision floating-point values from the second source operand and the first source operand, and stores the double precision floating-point result in the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source and destination operands are the same. Bits (MAXVL-1:32) of the corresponding the destination register remain unchanged.

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. 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 version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VADDSS is encoded with VEX.L=0. Encoding VADDSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VADDSS (EVEX Encoded Versions)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC1[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] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VADDSS DEST, SRC1, SRC2 (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

ADDSS DEST, SRC (128-bit Legacy SSE Version)

```
DEST[31:0] := DEST[31:0] + SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VADDSS __m128 _mm_mask_add_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_maskz_add_ss (__mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_add_round_ss (__m128 a, __m128 b, int);
VADDSS __m128 _mm_mask_add_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VADDSS __m128 _mm_maskz_add_round_ss (__mmask8 k, __m128 a, __m128 b, int);
ADDSS __m128 _mm_add_ss (__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

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

AESDEC—Perform One Round of an AES Decryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DE /r AESDEC xmm1, xmm2/m128	A	V/V	AES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DE /r VAESDEC ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128	C	V/V	VAES (AVX512VL OR AVX10.1[1])	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DE /r VAESDEC ymm1, ymm2, ymm3/m256	C	V/V	VAES (AVX512VL OR AVX10.1[1])	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DE /r VAESDEC zmm1, zmm2, zmm3/m512	C	V/V	VAES (AVX512F OR AVX10.1[1])	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using four 128-bit data (state) from zmm2 with four 128-bit round keys from zmm3/m512; store the result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a single round of the AES decryption flow using the Equivalent Inverse Cipher, using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

Use the AESDEC instruction for all but the last decryption round. For the last decryption round, use the AESDECLAST instruction.

VEX and EVEX encoded versions of the instruction allow 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESDEC

```
STATE := SRC1;
RoundKey := SRC2;
STATE := InvShiftRows( STATE );
STATE := InvSubBytes( STATE );
STATE := InvMixColumns( STATE );
DEST[127:0] := STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
```

VAESDEC (128b and 256b VEX Encoded Versions)

```
(KL,VL) = (1,128), (2,256)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    STATE := InvMixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

VAESDEC (EVEX Encoded Version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    STATE := InvMixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] :=0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)AESDEC __m128i _mm_aesdec (__m128i, __m128i)
VAESDEC __m256i _mm256_aesdec_epi128(__m256i, __m256i);
VAESDEC __m512i _mm512_aesdec_epi128(__m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

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

AESDECLAST—Perform Last Round of an AES Decryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DF /r AESDECLAST xmm1, xmm2/m128	A	V/V	AES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DF /r VAESDECLAST ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128	C	V/V	VAES (AVX512VL OR AVX10.1[1])	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DF /r VAESDECLAST ymm1, ymm2, ymm3/m256	C	V/V	VAES (AVX512VL OR AVX10.1[1])	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DF /r VAESDECLAST zmm1, zmm2, zmm3/m512	C	V/V	VAES (AVX512F OR AVX10.1[1])	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using four128-bit data (state) from zmm2 with four 128-bit round keys from zmm3/m512; store the result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs the last round of the AES decryption flow using the Equivalent Inverse Cipher, using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

VEX and EVEX encoded versions of the instruction allow 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESDECLAST

```
STATE := SRC1;
RoundKey := SRC2;
STATE := InvShiftRows( STATE );
STATE := InvSubBytes( STATE );
DEST[127:0] := STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
```

VAESDECLAST (128b and 256b VEX Encoded Versions)

```
(KL,VL) = (1,128), (2,256)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

VAESDECLAST (EVEX Encoded Version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)AESDECLAST __m128i _mm_aesdeclast (__m128i, __m128i)
VAESDECLAST __m256i _mm256_aesdeclast_epi128(__m256i, __m256i);
VAESDECLAST __m512i _mm512_aesdeclast_epi128(__m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

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

AESENC—Perform One Round of an AES Encryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DC /r AESENC xmm1, xmm2/m128	A	V/V	AES	Perform one round of an AES encryption flow, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DC /r VAESENC xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform one round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from the xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DC /r VAESENC ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform one round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from the ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DC /r VAESENC xmm1, xmm2, xmm3/m128	C	V/V	VAES (AVX512VL OR AVX10.1[1])	Perform one round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from the xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DC /r VAESENC ymm1, ymm2, ymm3/m256	C	V/V	VAES (AVX512VL OR AVX10.1[1])	Perform one round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from the ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DC /r VAESENC zmm1, zmm2, zmm3/m512	C	V/V	VAES (AVX512F OR AVX10.1[1])	Perform one round of an AES encryption flow, using four 128-bit data (state) from zmm2 with four 128-bit round keys from the zmm3/m512; store the result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a single round of an AES encryption flow using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

Use the AESENC instruction for all but the last encryption rounds. For the last encryption round, use the AESENCCLAST instruction.

VEX and EVEX encoded versions of the instruction allow 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESENC

```
STATE := SRC1;
RoundKey := SRC2;
STATE := ShiftRows( STATE );
STATE := SubBytes( STATE );
STATE := MixColumns( STATE );
DEST[127:0] := STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
```

VAESENC (128b and 256b VEX Encoded Versions)

```
(KL,VL) = (1,128), (2,256)
FOR I := 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := ShiftRows( STATE )
    STATE := SubBytes( STATE )
    STATE := MixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

VAESENC (EVEX Encoded Version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i := 0 to KL-1:
    STATE := SRC1.xmm[i] // xmm[i] is the i’th xmm word in the SIMD register
    RoundKey := SRC2.xmm[i]
    STATE := ShiftRows( STATE )
    STATE := SubBytes( STATE )
    STATE := MixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)AESENC __m128i _mm_aesenc (__m128i, __m128i)
VAESENC __m256i _mm256_aesenc_epi128(__m256i, __m256i);
VAESENC __m512i _mm512_aesenc_epi128(__m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

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

AESENCLAST—Perform Last Round of an AES Encryption Flow

Opcode/ Instruction	**Op/ En**	**64/32-bit Mode**	**CPUID Feature Flag**	**Description**
66 0F 38 DD /r AESENCLAST xmm1, xmm2/m128	A	V/V	AES	Perform the last round of an AES encryption flow, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DD /r VAESENCLAST xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform the last round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DD /r VAESENCLAST ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform the last round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DD /r VAESENCLAST xmm1, xmm2, xmm3/m128	C	V/V	VAES (AVX512VL OR AVX10.1[1])	Perform the last round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DD /r VAESENCLAST ymm1, ymm2, ymm3/m256	C	V/V	VAES (AVX512VL OR AVX10.1[1])	Perform the last round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DD /r VAESENCLAST zmm1, zmm2, zmm3/m512	C	V/V	VAES (AVX512F OR AVX10.1[1])	Perform the last round of an AES encryption flow, using four 128-bit data (state) from zmm2 with four 128-bit round keys from zmm3/m512; store the result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs the last round of an AES encryption flow using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

VEX and EVEX encoded versions of the instruction allows 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESENCLAST

```
STATE := SRC1;
RoundKey := SRC2;
STATE := ShiftRows( STATE );
STATE := SubBytes( STATE );
DEST[127:0] := STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
```

VAESENCLAST (128b and 256b VEX Encoded Versions)

```
(KL, VL) = (1,128), (2,256)
FOR I=0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := ShiftRows( STATE )
    STATE := SubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

VAESENCLAST (EVEX Encoded Version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := ShiftRows( STATE )
    STATE := SubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)AESENCLAST __m128i _mm_aesenclast (__m128i, __m128i)
VAESENCLAST __m256i _mm256_aesenclast_epi128(__m256i, __m256i);
VAESENCLAST __m512i _mm512_aesenclast_epi128(__m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

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

ANDPD—Bitwise Logical AND of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 54 /r ANDPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical AND of packed double precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F 54 /r VANDPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND of packed double precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F 54 /r VANDPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND of packed double precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 54 /r VANDPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Return the bitwise logical AND of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 54 /r VANDPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Return the bitwise logical AND of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 54 /r VANDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ OR AVX10.1[1]	Return the bitwise logical AND of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a bitwise logical AND 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 encoded versions: 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.

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. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: 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. 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

VANDPD (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 AND SRC2[63:0]
                ELSE
                    DEST[i+63:i] := SRC1[i+63:i] BITWISE AND 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
```

VANDPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE AND SRC2[127:64]
DEST[191:128] := SRC1[191:128] BITWISE AND SRC2[191:128]
DEST[255:192] := SRC1[255:192] BITWISE AND SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VANDPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE AND SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

ANDPD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] BITWISE AND SRC[63:0]
DEST[127:64] := DEST[127:64] BITWISE AND SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VANDPD __m512d _mm512_and_pd (__m512d a, __m512d b);
VANDPD __m512d _mm512_mask_and_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VANDPD __m512d _mm512_maskz_and_pd (__mmask8 k, __m512d a, __m512d b);
VANDPD __m256d _mm256_mask_and_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VANDPD __m256d _mm256_maskz_and_pd (__mmask8 k, __m256d a, __m256d b);
VANDPD __m128d _mm_mask_and_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VANDPD __m128d _mm_maskz_and_pd (__mmask8 k, __m128d a, __m128d b);
VANDPD __m256d _mm256_and_pd (__m256d a, __m256d b);
ANDPD __m128d _mm_and_pd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

ANDPS—Bitwise Logical AND of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 54 /r ANDPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical AND of packed single precision floating-point values in xmm1 and xmm2/mem.
VEX.128.0F 54 /r VANDPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND of packed single precision floating-point values in xmm2 and xmm3/mem.
VEX.256.0F 54 /r VANDPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND of packed single precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.0F.W0 54 /r VANDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Return the bitwise logical AND of packed single precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.0F.W0 54 /r VANDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Return the bitwise logical AND of packed single precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.0F.W0 54 /r VANDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ OR AVX10.1[1]	Return the bitwise logical AND of packed single precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a bitwise logical AND 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 encoded versions: 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-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: 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. 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 ZMM register destination are unmodified.

Operation

VANDPS (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*
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := SRC1[i+31:i] BITWISE AND SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC1[i+31:i] BITWISE AND 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;
```

VANDPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE AND SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE AND SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE AND SRC2[127:96]
DEST[159:128] := SRC1[159:128] BITWISE AND SRC2[159:128]
DEST[191:160] := SRC1[191:160] BITWISE AND SRC2[191:160]
DEST[223:192] := SRC1[223:192] BITWISE AND SRC2[223:192]
DEST[255:224] := SRC1[255:224] BITWISE AND SRC2[255:224].
DEST[MAXVL-1:256] := 0;
```

VANDPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE AND SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE AND SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE AND SRC2[127:96]
DEST[MAXVL-1:128] := 0;
```

ANDPS (128-bit Legacy SSE Version)

```
DEST[31:0] := DEST[31:0] BITWISE AND SRC[31:0]
DEST[63:32] := DEST[63:32] BITWISE AND SRC[63:32]
DEST[95:64] := DEST[95:64] BITWISE AND SRC[95:64]
DEST[127:96] := DEST[127:96] BITWISE AND SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VANDPS __m512 _mm512_and_ps (__m512 a, __m512 b);
VANDPS __m512 _mm512_mask_and_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDPS __m512 _mm512_maskz_and_ps (__mmask16 k, __m512 a, __m512 b);
VANDPS __m256 _mm256_mask_and_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDPS __m256 _mm256_maskz_and_ps (__mmask8 k, __m256 a, __m256 b);
VANDPS __m128 _mm_mask_and_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDPS __m128 _mm_maskz_and_ps (__mmask8 k, __m128 a, __m128 b);
VANDPS __m256 _mm256_and_ps (__m256 a, __m256 b);
ANDPS __m128 _mm_and_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

ANDNPD—Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 55 /r ANDNPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical AND NOT of packed double precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F 55 /r VANDNPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND NOT of packed double precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F 55/r VANDNPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND NOT of packed double precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 55 /r VANDNPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Return the bitwise logical AND NOT of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 55 /r VANDNPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Return the bitwise logical AND NOT of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 55 /r VANDNPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ OR AVX10.1[1]	Return the bitwise logical AND NOT of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a bitwise logical AND NOT 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 encoded versions: 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.

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. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: 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. 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

VANDNPD (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*
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := (NOT(SRC1[i+63:i])) BITWISE AND SRC2[63:0]
                ELSE
                    DEST[i+63:i] := (NOT(SRC1[i+63:i])) BITWISE AND 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
```

VANDNPD (VEX.256 Encoded Version)

```
DEST[63:0] := (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]
DEST[127:64] := (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]
DEST[191:128] := (NOT(SRC1[191:128])) BITWISE AND SRC2[191:128]
DEST[255:192] := (NOT(SRC1[255:192])) BITWISE AND SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VANDNPD (VEX.128 Encoded Version)

```
DEST[63:0] := (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]
DEST[127:64] := (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

ANDNPD (128-bit Legacy SSE Version)

```
DEST[63:0] := (NOT(DEST[63:0])) BITWISE AND SRC[63:0]
DEST[127:64] := (NOT(DEST[127:64])) BITWISE AND SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VANDNPD __m512d _mm512_andnot_pd (__m512d a, __m512d b);
VANDNPD __m512d _mm512_mask_andnot_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VANDNPD __m512d _mm512_maskz_andnot_pd (__mmask8 k, __m512d a, __m512d b);
VANDNPD __m256d _mm256_mask_andnot_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VANDNPD __m256d _mm256_maskz_andnot_pd (__mmask8 k, __m256d a, __m256d b);
VANDNPD __m128d _mm_mask_andnot_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VANDNPD __m128d _mm_maskz_andnot_pd (__mmask8 k, __m128d a, __m128d b);
VANDNPD __m256d _mm256_andnot_pd (__m256d a, __m256d b);
ANDNPD __m128d _mm_andnot_pd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

ANDNPS—Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 55 /r ANDNPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical AND NOT of packed single precision floating-point values in xmm1 and xmm2/mem.
VEX.128.0F 55 /r VANDNPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND NOT of packed single precision floating-point values in xmm2 and xmm3/mem.
VEX.256.0F 55 /r VANDNPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND NOT of packed single precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.0F.W0 55 /r VANDNPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Return the bitwise logical AND of packed single precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.0F.W0 55 /r VANDNPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Return the bitwise logical AND of packed single precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.0F.W0 55 /r VANDNPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ OR AVX10.1[1]	Return the bitwise logical AND of packed single precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a bitwise logical AND NOT 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 encoded versions: 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-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: 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. 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 ZMM register destination are unmodified.

Operation

VANDNPS (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*
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := (NOT(SRC1[i+31:i])) BITWISE AND SRC2[31:0]
                ELSE
                    DEST[i+31:i] := (NOT(SRC1[i+31:i])) BITWISE AND 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
```

VANDNPS (VEX.256 Encoded Version)

```
DEST[31:0] := (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[63:32] := (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]
DEST[95:64] := (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] := (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[159:128] := (NOT(SRC1[159:128])) BITWISE AND SRC2[159:128]
DEST[191:160] := (NOT(SRC1[191:160])) BITWISE AND SRC2[191:160]
DEST[223:192] := (NOT(SRC1[223:192])) BITWISE AND SRC2[223:192]
DEST[255:224] := (NOT(SRC1[255:224])) BITWISE AND SRC2[255:224].
DEST[MAXVL-1:256] := 0
```

VANDNPS (VEX.128 Encoded Version)

```
DEST[31:0] := (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[63:32] := (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]
DEST[95:64] := (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] := (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

ANDNPS (128-bit Legacy SSE Version)

```
DEST[31:0] := (NOT(DEST[31:0])) BITWISE AND SRC[31:0]
DEST[63:32] := (NOT(DEST[63:32])) BITWISE AND SRC[63:32]
DEST[95:64] := (NOT(DEST[95:64])) BITWISE AND SRC[95:64]
DEST[127:96] := (NOT(DEST[127:96])) BITWISE AND SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VANDNPS __m512 _mm512_andnot_ps (__m512 a, __m512 b);
VANDNPS __m512 _mm512_mask_andnot_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDNPS __m512 _mm512_maskz_andnot_ps (__mmask16 k, __m512 a, __m512 b);
VANDNPS __m256 _mm256_mask_andnot_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDNPS __m256 _mm256_maskz_andnot_ps (__mmask8 k, __m256 a, __m256 b);
VANDNPS __m128 _mm_mask_andnot_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDNPS __m128 _mm_maskz_andnot_ps (__mmask8 k, __m128 a, __m128 b);
VANDNPS __m256 _mm256_andnot_ps (__m256 a, __m256 b);
ANDNPS __m128 _mm_andnot_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

CMPPD—Compare Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C2 /r ib CMPPD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Compare packed double precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEX.128.66.0F.WIG C2 /r ib VCMPPD xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Compare packed double precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEX.256.66.0F.WIG C2 /r ib VCMPPD ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Compare packed double precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.
EVEX.128.66.0F.W1 C2 /r ib VCMPPD k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed double precision floating-point values in xmm3/m128/m64bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F.W1 C2 /r ib VCMPPD k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed double precision floating-point values in ymm3/m256/m64bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F.W1 C2 /r ib VCMPPD k1 {k2}, zmm2, zmm3/m512/m64bcst{sae}, imm8	C	V/V	AVX512F OR AVX10.1[1]	Compare packed double precision floating-point values in zmm3/m512/m64bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed double precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the 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.

EVEX encoded versions: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Four comparisons are performed with results written to the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Two comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. Two comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX or EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 1-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 1-1). Bits 3 through 7 of the immediate are reserved.

Table 1-1. Comparison Predicate for CMPPD and CMPPS Instructions

Predicate	**imm8 Value**	**Description**	**Result: A Is 1st Operand, B Is 2nd Operand**				**Signals #IA on QNAN**
			A >B	**A < B**	**A = B**	**Unordered[1]**	
EQ_OQ (EQ)	0H	Equal (ordered, non-signaling)	False	False	True	False	No
LT_OS (LT)	1H	Less-than (ordered, signaling)	False	True	False	False	Yes
LE_OS (LE)	2H	Less-than-or-equal (ordered, signaling)	False	True	True	False	Yes
UNORD_Q (UNORD)	3H	Unordered (non-signaling)	False	False	False	True	No
NEQ_UQ (NEQ)	4H	Not-equal (unordered, non-signaling)	True	True	False	True	No
NLT_US (NLT)	5H	Not-less-than (unordered, signaling)	True	False	True	True	Yes
NLE_US (NLE)	6H	Not-less-than-or-equal (unordered, signaling)	True	False	False	True	Yes
ORD_Q (ORD)	7H	Ordered (non-signaling)	True	True	True	False	No
EQ_UQ	8H	Equal (unordered, non-signaling)	False	False	True	True	No
NGE_US (NGE)	9H	Not-greater-than-or-equal (unordered, signaling)	False	True	False	True	Yes
NGT_US (NGT)	AH	Not-greater-than (unordered, signaling)	False	True	True	True	Yes
FALSE_OQ(FALSE)	BH	False (ordered, non-signaling)	False	False	False	False	No
NEQ_OQ	CH	Not-equal (ordered, non-signaling)	True	True	False	False	No
GE_OS (GE)	DH	Greater-than-or-equal (ordered, signaling)	True	False	True	False	Yes
GT_OS (GT)	EH	Greater-than (ordered, signaling)	True	False	False	False	Yes
TRUE_UQ(TRUE)	FH	True (unordered, non-signaling)	True	True	True	True	No
EQ_OS	10H	Equal (ordered, signaling)	False	False	True	False	Yes
LT_OQ	11H	Less-than (ordered, nonsignaling)	False	True	False	False	No
LE_OQ	12H	Less-than-or-equal (ordered, nonsignaling)	False	True	True	False	No
UNORD_S	13H	Unordered (signaling)	False	False	False	True	Yes
NEQ_US	14H	Not-equal (unordered, signaling)	True	True	False	True	Yes
NLT_UQ	15H	Not-less-than (unordered, nonsignaling)	True	False	True	True	No
NLE_UQ	16H	Not-less-than-or-equal (unordered, nonsignaling)	True	False	False	True	No
ORD_S	17H	Ordered (signaling)	True	True	True	False	Yes
EQ_US	18H	Equal (unordered, signaling)	False	False	True	True	Yes

Table 1-1. Comparison Predicate for CMPPD and CMPPS Instructions (Contd.)

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNAN
			A >B	A < B	A = B	Unordered[1]	
NGE_UQ	19H	Not-greater-than-or-equal (unordered, non-signaling)	False	True	False	True	No
NGT_UQ	1AH	Not-greater-than (unordered, nonsignaling)	False	True	True	True	No
FALSE_OS	1BH	False (ordered, signaling)	False	False	False	False	Yes
NEQ_OS	1CH	Not-equal (ordered, signaling)	True	True	False	False	Yes
GE_OQ	1DH	Greater-than-or-equal (ordered, nonsignaling)	True	False	True	False	No
GT_OQ	1EH	Greater-than (ordered, nonsignaling)	True	False	False	False	No
TRUE_US	1FH	True (unordered, signaling)	True	True	True	True	Yes

NOTES:

1. If either operand A or B is a NAN.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 1-2. The compiler should treat reserved imm8 values as illegal syntax.

Table 1-2. Pseudo-Op and CMPPD Implementation

Pseudo-Op	CMPPD Implementation
CMPEQPD xmm1, xmm2	CMPPD xmm1, xmm2, 0
CMPLTPD xmm1, xmm2	CMPPD xmm1, xmm2, 1
CMPLEPD xmm1, xmm2	CMPPD xmm1, xmm2, 2
CMPUNORDPD xmm1, xmm2	CMPPD xmm1, xmm2, 3
CMPNEQPD xmm1, xmm2	CMPPD xmm1, xmm2, 4
CMPNLTPD xmm1, xmm2	CMPPD xmm1, xmm2, 5
CMPNLEPD xmm1, xmm2	CMPPD xmm1, xmm2, 6
CMPORDPD xmm1, xmm2	CMPPD xmm1, xmm2, 7

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 1-3, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPD instruction. See Table 1-3, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPD instructions in a similar fashion by extending the syntax listed in Table 1-3.

Table 1-3. Pseudo-Op and VCMPPD Implementation

Pseudo-Op	**CMPPD Implementation**
VCMPEQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0*
VCMPLTPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1*
VCMPLEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 2*
VCMPUNORDPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 3*
VCMPNEQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 4*
VCMPNLTPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 5*
VCMPNLEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 6*
VCMPORDPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 7*
VCMPEQ_UQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 8*
VCMPNGEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 9*
VCMPNGTPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0AH*
VCMPFALSEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0BH*
VCMPNEQ_OQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0CH*
VCMPGEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0DH*
VCMPGTPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0EH*
VCMPTRUEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0FH*
VCMPEQ_OSPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 10H*
VCMPLT_OQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 11H*
VCMPLE_OQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 12H*
VCMPUNORD_SPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 13H*
VCMPNEQ_USPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 14H*
VCMPNLT_UQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 15H*
VCMPNLE_UQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 16H*
VCMPORD_SPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 17H*
VCMPEQ_USPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 18H*
VCMPNGE_UQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 19H*
VCMPNGT_UQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1AH*
VCMPFALSE_OSPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1BH*
VCMPNEQ_OSPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1CH*
VCMPGE_OQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1DH*
VCMPGT_OQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1EH*
VCMPTRUE_USPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1FH*

Operation

```
CASE (COMPARISON PREDICATE) OF
0: OP3 := EQ_OQ; OP5 := EQ_OQ;
    1: OP3 := LT_OS; OP5 := LT_OS;
    2: OP3 := LE_OS; OP5 := LE_OS;
    3: OP3 := UNORD_Q; OP5 := UNORD_Q;
    4: OP3 := NEQ_UQ; OP5 := NEQ_UQ;
    5: OP3 := NLT_US; OP5 := NLT_US;
    6: OP3 := NLE_US; OP5 := NLE_US;
    7: OP3 := ORD_Q; OP5 := ORD_Q;
    8: OP5 := EQ_UQ;
    9: OP5 := NGE_US;
    10: OP5 := NGT_US;
    11: OP5 := FALSE_OQ;
    12: OP5 := NEQ_OQ;
    13: OP5 := GE_OS;
    14: OP5 := GT_OS;
    15: OP5 := TRUE_UQ;
    16: OP5 := EQ_OS;
    17: OP5 := LT_OQ;
    18: OP5 := LE_OQ;
    19: OP5 := UNORD_S;
    20: OP5 := NEQ_US;
    21: OP5 := NLT_UQ;
    22: OP5 := NLE_UQ;
    23: OP5 := ORD_S;
    24: OP5 := EQ_US;
    25: OP5 := NGE_UQ;
    26: OP5 := NGT_UQ;
    27: OP5 := FALSE_OS;
    28: OP5 := NEQ_OS;
    29: OP5 := GE_OQ;
    30: OP5 := GT_OQ;
    31: OP5 := TRUE_US;
    DEFAULT: Reserved;
ESAC;
```

VCMPPD (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] OP5 SRC2[63:0]
                ELSE
                    CMP := SRC1[i+63:i] OP5 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
```

VCMPPD (VEX.256 Encoded Version)

```
CMP0 := SRC1[63:0] OP5 SRC2[63:0];
CMP1 := SRC1[127:64] OP5 SRC2[127:64];
CMP2 := SRC1[191:128] OP5 SRC2[191:128];
CMP3 := SRC1[255:192] OP5 SRC2[255:192];
IF CMP0 = TRUE
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0000000000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[191:128] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[191:128] := 0000000000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[255:192] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[255:192] := 0000000000000000H; FI;
DEST[MAXVL-1:256] := 0
```

VCMPPD (VEX.128 Encoded Version)

```
CMP0 := SRC1[63:0] OP5 SRC2[63:0];
CMP1 := SRC1[127:64] OP5 SRC2[127:64];
IF CMP0 = TRUE
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0000000000000000H; FI;
DEST[MAXVL-1:128] := 0
```

CMPPD (128-bit Legacy SSE Version)

```
CMP0 := SRC1[63:0] OP3 SRC2[63:0];
CMP1 := SRC1[127:64] OP3 SRC2[127:64];
IF CMP0 = TRUE
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0000000000000000H; FI;
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCMPPD __mmask8 _mm512_cmp_pd_mask( __m512d a, __m512d b, int imm);
VCMPPD __mmask8 _mm512_cmp_round_pd_mask( __m512d a, __m512d b, int imm, int sae);
VCMPPD __mmask8 _mm512_mask_cmp_pd_mask( __mmask8 k1, __m512d a, __m512d b, int imm);
VCMPPD __mmask8 _mm512_mask_cmp_round_pd_mask( __mmask8 k1, __m512d a, __m512d b, int imm, int sae);
VCMPPD __mmask8 _mm256_cmp_pd_mask( __m256d a, __m256d b, int imm);
VCMPPD __mmask8 _mm256_mask_cmp_pd_mask( __mmask8 k1, __m256d a, __m256d b, int imm);
VCMPPD __mmask8 _mm_cmp_pd_mask( __m128d a, __m128d b, int imm);
VCMPPD __mmask8 _mm_mask_cmp_pd_mask( __mmask8 k1, __m128d a, __m128d b, int imm);
VCMPPD __m256 _mm256_cmp_pd(__m256d a, __m256d b, int imm)
```

(V)CMPPD __m128 _mm_cmp_pd(__m128d a, __m128d b, int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 1-1.

Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."

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

CMPPS—Compare Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C2 /r ib CMPPS xmm1, xmm2/m128, imm8	A	V/V	SSE	Compare packed single precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEX.128.0F.WIG C2 /r ib VCMPPS xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Compare packed single precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEX.256.0F.WIG C2 /r ib VCMPPS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Compare packed single precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.
EVEX.128.0F.W0 C2 /r ib VCMPPS k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed single precision floating-point values in xmm3/m128/m32bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.0F.W0 C2 /r ib VCMPPS k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed single precision floating-point values in ymm3/m256/m32bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.0F.W0 C2 /r ib VCMPPS k1 {k2}, zmm2, zmm3/m512/m32bcst{sae}, imm8	C	V/V	AVX512F OR AVX10.1[1]	Compare packed single precision floating-point values in zmm3/m512/m32bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed single precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each of the pairs of packed values.

EVEX encoded versions: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Eight comparisons are performed with results written to the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. Four comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix and EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 1-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 1-1). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 1-4. The compiler should treat reserved imm8 values as illegal syntax.

Table 1-4. Pseudo-Op and CMPPS Implementation

Pseudo-Op	CMPPS Implementation
CMPEQPS xmm1, xmm2	CMPPS xmm1, xmm2, 0
CMPLTPS xmm1, xmm2	CMPPS xmm1, xmm2, 1
CMPLEPS xmm1, xmm2	CMPPS xmm1, xmm2, 2
CMPUNORDPS xmm1, xmm2	CMPPS xmm1, xmm2, 3
CMPNEQPS xmm1, xmm2	CMPPS xmm1, xmm2, 4
CMPNLTPS xmm1, xmm2	CMPPS xmm1, xmm2, 5
CMPNLEPS xmm1, xmm2	CMPPS xmm1, xmm2, 6
CMPORDPS xmm1, xmm2	CMPPS xmm1, xmm2, 7

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 1-5, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPS instruction. See Table 1-5, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPS instructions in a similar fashion by extending the syntax listed in Table 1-5.

Table 1-5. Pseudo-Op and VCMPPS Implementation

Pseudo-Op	**CMPPS Implementation**
VCMPEQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0*
VCMPLTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1*
VCMPLEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 2*
VCMPUNORDPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 3*
VCMPNEQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 4*
VCMPNLTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 5*
VCMPNLEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 6*
VCMPORDPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 7*
VCMPEQ_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 8*
VCMPNGEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 9*
VCMPNGTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0AH*
VCMPFALSEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0BH*
VCMPNEQ_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0CH*
VCMPGEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0DH*
VCMPGTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0EH*
VCMPTRUEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0FH*
VCMPEQ_OSPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 10H*
VCMPLT_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 11H*
VCMPLE_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 12H*
VCMPUNORD_SPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 13H*
VCMPNEQ_USPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 14H*
VCMPNLT_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 15H*
VCMPNLE_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 16H*
VCMPORD_SPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 17H*
VCMPEQ_USPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 18H*
VCMPNGE_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 19H*
VCMPNGT_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1AH*
VCMPFALSE_OSPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1BH*
VCMPNEQ_OSPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1CH*
VCMPGE_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1DH*
VCMPGT_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1EH*
VCMPTRUE_USPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1FH*

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP3 := EQ_OQ; OP5 := EQ_OQ;
    1: OP3 := LT_OS; OP5 := LT_OS;
    2: OP3 := LE_OS; OP5 := LE_OS;
    3: OP3 := UNORD_Q; OP5 := UNORD_Q;
    4: OP3 := NEQ_UQ; OP5 := NEQ_UQ;
    5: OP3 := NLT_US; OP5 := NLT_US;
    6: OP3 := NLE_US; OP5 := NLE_US;
    7: OP3 := ORD_Q; OP5 := ORD_Q;
    8: OP5 := EQ_UQ;
    9: OP5 := NGE_US;
    10: OP5 := NGT_US;
    11: OP5 := FALSE_OQ;
    12: OP5 := NEQ_OQ;
    13: OP5 := GE_OS;
    14: OP5 := GT_OS;
    15: OP5 := TRUE_UQ;
    16: OP5 := EQ_OS;
    17: OP5 := LT_OQ;
    18: OP5 := LE_OQ;
    19: OP5 := UNORD_S;
    20: OP5 := NEQ_US;
    21: OP5 := NLT_UQ;
    22: OP5 := NLE_UQ;
    23: OP5 := ORD_S;
    24: OP5 := EQ_US;
    25: OP5 := NGE_UQ;
    26: OP5 := NGT_UQ;
    27: OP5 := FALSE_OS;
    28: OP5 := NEQ_OS;
    29: OP5 := GE_OQ;
    30: OP5 := GT_OQ;
    31: OP5 := TRUE_US;
    DEFAULT: Reserved
ESAC;
```

VCMPPS (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] OP5 SRC2[31:0]
                ELSE
                    CMP := SRC1[i+31:i] OP5 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
```

VCMPPS (VEX.256 Encoded Version)

```
CMP0 := SRC1[31:0] OP5 SRC2[31:0];
CMP1 := SRC1[63:32] OP5 SRC2[63:32];
CMP2 := SRC1[95:64] OP5 SRC2[95:64];
CMP3 := SRC1[127:96] OP5 SRC2[127:96];
CMP4 := SRC1[159:128] OP5 SRC2[159:128];
CMP5 := SRC1[191:160] OP5 SRC2[191:160];
CMP6 := SRC1[223:192] OP5 SRC2[223:192];
CMP7 := SRC1[255:224] OP5 SRC2[255:224];
IF CMP0 = TRUE
    THEN DEST[31:0] :=FFFFFFFFH;
    ELSE DEST[31:0] := 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] := FFFFFFFFH;
    ELSE DEST[63:32] :=000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] := FFFFFFFFH;
    ELSE DEST[95:64] := 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] := FFFFFFFFH;
    ELSE DEST[127:96] := 000000000H; FI;
IF CMP4 = TRUE
    THEN DEST[159:128] := FFFFFFFFH;
    ELSE DEST[159:128] := 000000000H; FI;
IF CMP5 = TRUE
    THEN DEST[191:160] := FFFFFFFFH;
    ELSE DEST[191:160] := 000000000H; FI;
IF CMP6 = TRUE
    THEN DEST[223:192] := FFFFFFFFH;
    ELSE DEST[223:192] :=000000000H; FI;
IF CMP7 = TRUE
    THEN DEST[255:224] := FFFFFFFFH;
    ELSE DEST[255:224] := 000000000H; FI;
DEST[MAXVL-1:256] := 0
```

VCMPPS (VEX.128 Encoded Version)

```
CMP0 := SRC1[31:0] OP5 SRC2[31:0];
CMP1 := SRC1[63:32] OP5 SRC2[63:32];
CMP2 := SRC1[95:64] OP5 SRC2[95:64];
CMP3 := SRC1[127:96] OP5 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] :=FFFFFFFFH;
    ELSE DEST[31:0] := 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] := FFFFFFFFH;
    ELSE DEST[63:32] := 000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] := FFFFFFFFH;
    ELSE DEST[95:64] := 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] := FFFFFFFFH;
```

```
    ELSE DEST[127:96] :=000000000H; FI;
DEST[MAXVL-1:128] := 0
```

CMPPS (128-bit Legacy SSE Version)

```
CMP0 := SRC1[31:0] OP3 SRC2[31:0];
CMP1 := SRC1[63:32] OP3 SRC2[63:32];
CMP2 := SRC1[95:64] OP3 SRC2[95:64];
CMP3 := SRC1[127:96] OP3 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] :=FFFFFFFFH;
    ELSE DEST[31:0] := 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] := FFFFFFFFH;
    ELSE DEST[63:32] := 000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] := FFFFFFFFH;
    ELSE DEST[95:64] := 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] := FFFFFFFFH;
    ELSE DEST[127:96] :=000000000H; FI;
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCMPPS __mmask16 _mm512_cmp_ps_mask( __m512 a, __m512 b, int imm);
VCMPPS __mmask16 _mm512_cmp_round_ps_mask( __m512 a, __m512 b, int imm, int sae);
VCMPPS __mmask16 _mm512_mask_cmp_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm);
VCMPPS __mmask16 _mm512_mask_cmp_round_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm, int sae);
VCMPPS __mmask8 _mm256_cmp_ps_mask( __m256 a, __m256 b, int imm);
VCMPPS __mmask8 _mm256_mask_cmp_ps_mask( __mmask8 k1, __m256 a, __m256 b, int imm);
VCMPPS __mmask8 _mm_cmp_ps_mask( __m128 a, __m128 b, int imm);
VCMPPS __mmask8 _mm_mask_cmp_ps_mask( __mmask8 k1, __m128 a, __m128 b, int imm);
VCMPPS __m256 _mm256_cmp_ps(__m256 a, __m256 b, int imm)
CMPPS __m128 _mm_cmp_ps(__m128 a, __m128 b, int imm)
```

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 1-1.

Denormal.

Other Exceptions

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

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

CMPSD—Compare Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F C2 /r ib CMPSD xmm1, xmm2/m64, imm8	A	V/V	SSE2	Compare low double precision floating-point value in xmm2/m64 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.LIG.F2.0F.WIG C2 /r ib VCMPSD xmm1, xmm2, xmm3/m64, imm8	B	V/V	AVX	Compare low double precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.LLIG.F2.0F.W1 C2 /r ib VCMPSD k1 {k2}, xmm2, xmm3/m64{sae}, imm8	C	V/V	AVX512F OR AVX10.1[1]	Compare low double precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Compares the low double precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 64-bit memory location. Bits (MAXVL-1:64) of the corresponding YMM destination register remain unchanged. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 64-bit memory location. The result is stored in the low quadword of the destination operand; the high quadword is filled with the contents of the high quadword of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 64-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 1-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 1-1). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 1-6. The compiler should treat reserved imm8 values as illegal syntax.

Table 1-6. Pseudo-Op and CMPSD Implementation

Pseudo-Op	CMPSD Implementation
CMPEQSD xmm1, xmm2	CMPSD xmm1, xmm2, 0
CMPLTSD xmm1, xmm2	CMPSD xmm1, xmm2, 1
CMPLESD xmm1, xmm2	CMPSD xmm1, xmm2, 2
CMPUNORDSD xmm1, xmm2	CMPSD xmm1, xmm2, 3
CMPNEQSD xmm1, xmm2	CMPSD xmm1, xmm2, 4
CMPNLTSD xmm1, xmm2	CMPSD xmm1, xmm2, 5
CMPNLESD xmm1, xmm2	CMPSD xmm1, xmm2, 6
CMPORDSD xmm1, xmm2	CMPSD xmm1, xmm2, 7

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 1-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 1-7, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSD instructions in a similar fashion by extending the syntax listed in Table 1-7.

Table 1-7. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
VCMPEQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0
VCMPLTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1
VCMPLESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 2
VCMPUNORDSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 3
VCMPNEQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 4
VCMPNLTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 5
VCMPNLESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 6
VCMPORDSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 7
VCMPEQ_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 8
VCMPNGESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 9

Table 1-7. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
VCMPNGTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0AH
VCMPFALSESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0BH
VCMPNEQ_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0CH
VCMPGESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0DH
VCMPGTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0EH
VCMPTRUESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0FH
VCMPEQ_OSSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 10H
VCMPLT_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 11H
VCMPLE_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 12H
VCMPUNORD_SSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 13H
VCMPNEQ_USSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 14H
VCMPNLT_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 15H
VCMPNLE_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 16H
VCMPORD_SSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 17H
VCMPEQ_USSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 18H
VCMPNGE_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 19H
VCMPNGT_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1AH
VCMPFALSE_OSSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1BH
VCMPNEQ_OSSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1CH
VCMPGE_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1DH
VCMPGT_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1EH
VCMPTRUE_USSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1FH

Software should ensure VCMPSD is encoded with VEX.L=0. Encoding VCMPSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP3 := EQ_OQ; OP5 := EQ_OQ;
    1: OP3 := LT_OS; OP5 := LT_OS;
    2: OP3 := LE_OS; OP5 := LE_OS;
    3: OP3 := UNORD_Q; OP5 := UNORD_Q;
    4: OP3 := NEQ_UQ; OP5 := NEQ_UQ;
    5: OP3 := NLT_US; OP5 := NLT_US;
    6: OP3 := NLE_US; OP5 := NLE_US;
    7: OP3 := ORD_Q; OP5 := ORD_Q;
    8: OP5 := EQ_UQ;
    9: OP5 := NGE_US;
    10: OP5 := NGT_US;
    11: OP5 := FALSE_OQ;
    12: OP5 := NEQ_OQ;
    13: OP5 := GE_OS;
    14: OP5 := GT_OS;
    15: OP5 := TRUE_UQ;
```

```
    16: OP5 := EQ_OS;
    17: OP5 := LT_OQ;
    18: OP5 := LE_OQ;
    19: OP5 := UNORD_S;
    20: OP5 := NEQ_US;
    21: OP5 := NLT_UQ;
    22: OP5 := NLE_UQ;
    23: OP5 := ORD_S;
    24: OP5 := EQ_US;
    25: OP5 := NGE_UQ;
    26: OP5 := NGT_UQ;
    27: OP5 := FALSE_OS;
    28: OP5 := NEQ_OS;
    29: OP5 := GE_OQ;
    30: OP5 := GT_OQ;
    31: OP5 := TRUE_US;
    DEFAULT: Reserved
ESAC;
```

VCMPSD (EVEX Encoded Version)

```
CMP0 := SRC1[63:0] OP5 SRC2[63:0];

IF k2[0] or *no writemask*
    THEN    IF CMP0 = TRUE
                THEN DEST[0] := 1;
                ELSE DEST[0] := 0; FI;
    ELSE    DEST[0] := 0                ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0
```

CMPSD (128-bit Legacy SSE Version)

```
CMP0 := DEST[63:0] OP3 SRC[63:0];
IF CMP0 = TRUE
THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] := 0000000000000000H; FI;
DEST[MAXVL-1:64] (Unmodified)
```

VCMPSD (VEX.128 Encoded Version)

```
CMP0 := SRC1[63:0] OP5 SRC2[63:0];
IF CMP0 = TRUE
THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] := 0000000000000000H; FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCMPSD __mmask8 _mm_cmp_sd_mask( __m128d a, __m128d b, int imm);
VCMPSD __mmask8 _mm_cmp_round_sd_mask( __m128d a, __m128d b, int imm, int sae);
VCMPSD __mmask8 _mm_mask_cmp_sd_mask( __mmask8 k1, __m128d a, __m128d b, int imm);
VCMPSD __mmask8 _mm_mask_cmp_round_sd_mask( __mmask8 k1, __m128d a, __m128d b, int imm, int sae);
(V)CMPSD __m128d _mm_cmp_sd(__m128d a, __m128d b, const int imm)
```

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 1-1, Denormal.

Other Exceptions

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

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

CMPSS—Compare Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F C2 /r ib CMPSS xmm1, xmm2/m32, imm8	A	V/V	SSE	Compare low single precision floating-point value in xmm2/m32 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.LIG.F3.0F.WIG C2 /r ib VCMPSS xmm1, xmm2, xmm3/m32, imm8	B	V/V	AVX	Compare low single precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.LLIG.F3.0F.W0 C2 /r ib VCMPSS k1 {k2}, xmm2, xmm3/m32{sae}, imm8	C	V/V	AVX512F OR AVX10.1[1]	Compare low single precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

NOTES:

[1] For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Compares the low single precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 32-bit memory location. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 32-bit memory location. The result is stored in the low 32 bits of the destination operand; bits 127:32 of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 32-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 1-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 1-1). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 1-8. The compiler should treat reserved imm8 values as illegal syntax.

Table 1-8. Pseudo-Op and CMPSS Implementation

Pseudo-Op	CMPSS Implementation
CMPEQSS xmm1, xmm2	CMPSS xmm1, xmm2, 0
CMPLTSS xmm1, xmm2	CMPSS xmm1, xmm2, 1
CMPLESS xmm1, xmm2	CMPSS xmm1, xmm2, 2
CMPUNORDSS xmm1, xmm2	CMPSS xmm1, xmm2, 3
CMPNEQSS xmm1, xmm2	CMPSS xmm1, xmm2, 4
CMPNLTSS xmm1, xmm2	CMPSS xmm1, xmm2, 5
CMPNLESS xmm1, xmm2	CMPSS xmm1, xmm2, 6
CMPORDSS xmm1, xmm2	CMPSS xmm1, xmm2, 7

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 1-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSS instruction. See Table 1-9, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSS instructions in a similar fashion by extending the syntax listed in Table 1-9.

Table 1-9. Pseudo-Op and VCMPSS Implementation

Pseudo-Op	CMPSS Implementation
VCMPEQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0
VCMPLTSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1
VCMPLESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 2
VCMPUNORDSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 3
VCMPNEQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 4
VCMPNLTSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 5
VCMPNLESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 6
VCMPORDSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 7
VCMPEQ_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 8
VCMPNGESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 9

Table 1-9. Pseudo-Op and VCMPSS Implementation

Pseudo-Op	**CMPSS Implementation**
VCMPNGTSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0AH
VCMPFALSESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0BH
VCMPNEQ_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0CH
VCMPGESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0DH
VCMPGTSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0EH
VCMPTRUESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0FH
VCMPEQ_OSSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 10H
VCMPLT_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 11H
VCMPLE_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 12H
VCMPUNORD_SSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 13H
VCMPNEQ_USSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 14H
VCMPNLT_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 15H
VCMPNLE_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 16H
VCMPORD_SSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 17H
VCMPEQ_USSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 18H
VCMPNGE_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 19H
VCMPNGT_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1AH
VCMPFALSE_OSSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1BH
VCMPNEQ_OSSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1CH
VCMPGE_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1DH
VCMPGT_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1EH
VCMPTRUE_USSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1FH

Software should ensure VCMPSS is encoded with VEX.L=0. Encoding VCMPSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP3 := EQ_OQ; OP5 := EQ_OQ;
    1: OP3 := LT_OS; OP5 := LT_OS;
    2: OP3 := LE_OS; OP5 := LE_OS;
    3: OP3 := UNORD_Q; OP5 := UNORD_Q;
    4: OP3 := NEQ_UQ; OP5 := NEQ_UQ;
    5: OP3 := NLT_US; OP5 := NLT_US;
    6: OP3 := NLE_US; OP5 := NLE_US;
    7: OP3 := ORD_Q; OP5 := ORD_Q;
    8: OP5 := EQ_UQ;
    9: OP5 := NGE_US;
    10: OP5 := NGT_US;
    11: OP5 := FALSE_OQ;
    12: OP5 := NEQ_OQ;
    13: OP5 := GE_OS;
    14: OP5 := GT_OS;
    15: OP5 := TRUE_UQ;
```

```
    16: OP5 := EQ_OS;
    17: OP5 := LT_OQ;
    18: OP5 := LE_OQ;
    19: OP5 := UNORD_S;
    20: OP5 := NEQ_US;
    21: OP5 := NLT_UQ;
    22: OP5 := NLE_UQ;
    23: OP5 := ORD_S;
    24: OP5 := EQ_US;
    25: OP5 := NGE_UQ;
    26: OP5 := NGT_UQ;
    27: OP5 := FALSE_OS;
    28: OP5 := NEQ_OS;
    29: OP5 := GE_OQ;
    30: OP5 := GT_OQ;
    31: OP5 := TRUE_US;
    DEFAULT: Reserved
ESAC;
```

VCMPSS (EVEX Encoded Version)

```
CMP0 := SRC1[31:0] OP5 SRC2[31:0];

IF k2[0] or *no writemask*
    THEN    IF CMP0 = TRUE
                THEN DEST[0] := 1;
                ELSE DEST[0] := 0; FI;
    ELSE    DEST[0] := 0                ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0
```

CMPSS (128-bit Legacy SSE Version)

```
CMP0 := DEST[31:0] OP3 SRC[31:0];
IF CMP0 = TRUE
THEN DEST[31:0] := FFFFFFFFH;
ELSE DEST[31:0] := 00000000H; FI;
DEST[MAXVL-1:32] (Unmodified)
```

VCMPSS (VEX.128 Encoded Version)

```
CMP0 := SRC1[31:0] OP5 SRC2[31:0];
IF CMP0 = TRUE
THEN DEST[31:0] := FFFFFFFFH;
ELSE DEST[31:0] := 00000000H; FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCMPSS __mmask8 _mm_cmp_ss_mask(__m128 a, __m128 b, int imm);
VCMPSS __mmask8 _mm_cmp_round_ss_mask(__m128 a, __m128 b, int imm, int sae);
VCMPSS __mmask8 _mm_mask_cmp_ss_mask(__mmask8 k1, __m128 a, __m128 b, int imm);
VCMPSS __mmask8 _mm_mask_cmp_round_ss_mask(__mmask8 k1, __m128 a, __m128 b, int imm, int sae);
(V)CMPSS __m128 _mm_cmp_ss(__m128 a, __m128 b, const int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 1-1, Denormal.

Other Exceptions

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

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

COMISD—Compare Scalar Ordered Double Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2F /r COMISD xmm1, xmm2/m64	A	V/V	SSE2	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2F /r VCOMISD xmm1, xmm2/m64	A	V/V	AVX	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LLIG.66.0F.W1 2F /r VCOMISD xmm1, xmm2/m64{sae}	B	V/V	AVX512F OR AVX10.1[1]	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Compares the double precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory location. The COMISD instruction differs from the UCOMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISD instruction signals an invalid operation exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

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

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

COMISD (All Versions)

```
RESULT :=   OrderedCompare(DEST[63:0] <> SRC[63:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF := 111;
    GREATER_THAN: ZF,PF,CF := 000;
    LESS_THAN: ZF,PF,CF := 001;
    EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VCOMISD int _mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
VCOMISD int _mm_comieq_sd (__m128d a, __m128d b)
VCOMISD int _mm_comilt_sd (__m128d a, __m128d b)
VCOMISD int _mm_comile_sd (__m128d a, __m128d b)
VCOMISD int _mm_comigt_sd (__m128d a, __m128d b)
VCOMISD int _mm_comige_sd (__m128d a, __m128d b)
VCOMISD int _mm_comineq_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

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

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

Additionally:

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

COMISS—Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2F /r COMISS xmm1, xmm2/m32	A	V/V	SSE	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.0F.WIG 2F /r VCOMISS xmm1, xmm2/m32	A	V/V	AVX	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LLIG.0F.W0 2F /r VCOMISS xmm1, xmm2/m32{sae}	B	V/V	AVX512F OR AVX10.1[1]	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Compares the single precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

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

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

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

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

COMISS (All Versions)

```
RESULT :=    OrderedCompare(DEST[31:0] <> SRC[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF := 111;
    GREATER_THAN: ZF,PF,CF := 000;
    LESS_THAN: ZF,PF,CF := 001;
    EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VCOMISS int _mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
VCOMISS int _mm_comieq_ss (__m128 a, __m128 b)
VCOMISS int _mm_comilt_ss (__m128 a, __m128 b)
VCOMISS int _mm_comile_ss (__m128 a, __m128 b)
VCOMISS int _mm_comigt_ss (__m128 a, __m128 b)
VCOMISS int _mm_comige_ss (__m128 a, __m128 b)
VCOMISS int _mm_comineq_ss (__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

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

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

Additionally:

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

CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F E6 /r CVTDQ2PD xmm1, xmm2/m64	A	V/V	SSE2	Convert two packed signed doubleword integers from xmm2/mem to two packed double precision floating-point values in xmm1.
VEX.128.F3.0F.WIG E6 /r VCVTDQ2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed signed doubleword integers from xmm2/mem to two packed double precision floating-point values in xmm1.
VEX.256.F3.0F.WIG E6 /r VCVTDQ2PD ymm1, xmm2/m128	A	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed double precision floating-point values in ymm1.
EVEX.128.F3.0F.W0 E6 /r VCVTDQ2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert 2 packed signed doubleword integers from xmm2/m64/m32bcst to eight packed double precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W0 E6 /r VCVTDQ2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert 4 packed signed doubleword integers from xmm2/m128/m32bcst to 4 packed double precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W0 E6 /r VCVTDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512F OR AVX10.1[1]	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Converts two, four or eight packed signed doubleword integers in the source operand (the second operand) to two, four or eight packed double precision floating-point values in the destination operand (the first operand).

EVEX encoded versions: The source operand can be 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.

VEX.256 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a YMM register.

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

128-bit Legacy SSE version: The source operand is an XMM register or 64- bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

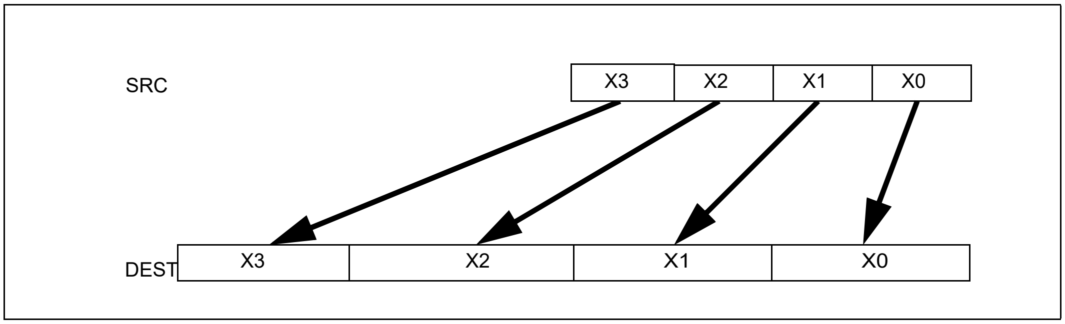


Figure 1-1. CVTDQ2PD (VEX.256 encoded version)

Operation

VCVTDQ2PD (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_Integer_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
```

VCVTDQ2PD (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_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Integer_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
```

VCVTDQ2PD (VEX.256 Encoded Version)

```
DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[127:96)
DEST[MAXVL-1:256] := 0
```

VCVTDQ2PD (VEX.128 Encoded Version)

```
DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] := 0
```

CVTDQ2PD (128-bit Legacy SSE Version)

```
DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTDQ2PD __m512d _mm512_cvtepi32_pd( __m256i a);
VCVTDQ2PD __m512d _mm512_mask_cvtepi32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m512d _mm512_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m256d _mm256_cvtepi32_pd (__m128i src);
VCVTDQ2PD __m256d _mm256_mask_cvtepi32_pd( __m256d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m256d _mm256_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m128d _mm_mask_cvtepi32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTDQ2PD __m128d _mm_maskz_cvtepi32_pd( __mmask8 k, __m128i a);
CVTDQ2PD __m128d _mm_cvtepi32_pd (__m128i src)
```

Other Exceptions

VEX-encoded instructions, see Table 2-22, “Type 5 Class Exception Conditions.”

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

Additionally:

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

CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single Precision Floating-Point Values

Opcode Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5B /r CVTDQ2PS xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed signed doubleword integers from xmm2/mem to four packed single precision floating-point values in xmm1.
VEX.128.0F.WIG 5B /r VCVTDQ2PS xmm1, xmm2/m128	A	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed single precision floating-point values in xmm1.
VEX.256.0F.WIG 5B /r VCVTDQ2PS ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed signed doubleword integers from ymm2/mem to eight packed single precision floating-point values in ymm1.
EVEX.128.0F.W0 5B /r VCVTDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed single precision floating-point values in xmm1with writemask k1.
EVEX.256.0F.W0 5B /r VCVTDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed single precision floating-point values in ymm1with writemask k1.
EVEX.512.0F.W0 5B /r VCVTDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed single precision floating-point values in zmm1with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Converts four, eight or sixteen packed signed doubleword integers in the source operand to four, eight or sixteen packed single precision floating-point values in the destination operand.

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

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

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

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.

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

Operation

VCVTDQ2PS (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);  ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures
Software Developer's Manual, Volume 1
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);  ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures
Software Developer's Manual, Volume 1
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Integer_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
```

VCVTDQ2PS (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_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Integer_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
```

VCVTDQ2PS (VEX.256 Encoded Version)

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

VCVTDQ2PS (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96)
DEST[MAXVL-1:128] := 0
```

CVTDQ2PS (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96)
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTDQ2PS __m512 _mm512_cvtepi32_ps( __m512i a);
VCVTDQ2PS __m512 _mm512_mask_cvtepi32_ps( __m512 s, __mmask16 k, __m512i a);
VCVTDQ2PS __m512 _mm512_maskz_cvtepi32_ps( __mmask16 k, __m512i a);
VCVTDQ2PS __m512 _mm512_cvt_roundepi32_ps( __m512i a, int r);
VCVTDQ2PS __m512 _mm512_mask_cvt_roundepi_ps( __m512 s, __mmask16 k, __m512i a, int r);
VCVTDQ2PS __m512 _mm512_maskz_cvt_roundepi32_ps( __mmask16 k, __m512i a, int r);
VCVTDQ2PS __m256 _mm256_mask_cvtepi32_ps( __m256 s, __mmask8 k, __m256i a);
VCVTDQ2PS __m256 _mm256_maskz_cvtepi32_ps( __mmask8 k, __m256i a);
VCVTDQ2PS __m128 _mm_mask_cvtepi32_ps( __m128 s, __mmask8 k, __m128i a);
VCVTDQ2PS __m128 _mm_maskz_cvtepi32_ps( __mmask8 k, __m128i a);
CVTDQ2PS __m256 _mm256_cvtepi32_ps (__m256i src)
CVTDQ2PS __m128 _mm_cvtepi32_ps (__m128i src)
```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."

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

Additionally:

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

CVTPD2DQ—Convert Packed Double Precision Floating-Point Values to Packed Doubleword Integers

Opcode Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F E6 /r CVTPD2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.
VEX.128.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.
VEX.256.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1.
EVEX.128.F2.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 subject to writemask k1.
EVEX.256.F2.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 subject to writemask k1.
EVEX.512.F2.0F.W1 E6 /r VCVTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Converts packed double precision floating-point values in the source operand (second operand) to packed signed doubleword 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 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 ZMM/YMM/XMM register, a 512-bit memory location, or a 512-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-1:256/128/64) of the corresponding destination are zeroed.

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

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

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

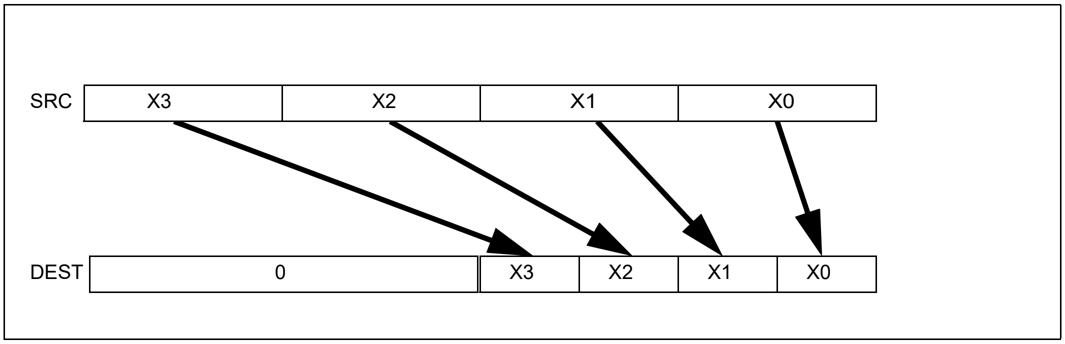


Figure 1-2. VCVTPD2DQ (VEX.256 Encoded Version)

Operation

VCVTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer(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
```

VCVTPD2DQ (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_Integer(SRC[63:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer(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
```

VCVTPD2DQ (VEX.256 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[95:64] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[191:128])
DEST[127:96] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[255:192)
DEST[MAXVL-1:128] := 0
```

VCVTPD2DQ (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[MAXVL-1:64] := 0
```

CVTPD2DQ (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[127:64] := 0
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2DQ __m256i _mm512_cvtpd_epi32(__m512d a);
VCVTPD2DQ __m256i _mm512_mask_cvtpd_epi32(__m256i s, __mmask8 k, __m512d a);
VCVTPD2DQ __m256i _mm512_maskz_cvtpd_epi32(__mmask8 k, __m512d a);
VCVTPD2DQ __m256i _mm512_cvt_roundpd_epi32(__m512d a, int r);
VCVTPD2DQ __m256i _mm512_mask_cvt_roundpd_epi32(__m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2DQ __m256i _mm512_maskz_cvt_roundpd_epi32(__mmask8 k, __m512d a, int r);
VCVTPD2DQ __m128i _mm256_mask_cvtpd_epi32(__m128i s, __mmask8 k, __m256d a);
VCVTPD2DQ __m128i _mm256_maskz_cvtpd_epi32(__mmask8 k, __m256d a);
VCVTPD2DQ __m128i _mm_mask_cvtpd_epi32(__m128i s, __mmask8 k, __m128d a);
VCVTPD2DQ __m128i _mm_maskz_cvtpd_epi32(__mmask8 k, __m128d a);
VCVTPD2DQ __m128i _mm256_cvtpd_epi32 (__m256d src)
CVTPD2DQ __m128i _mm_cvtpd_epi32 (__m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 2-19, “Type 2 Class Exception Conditions.”

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

Additionally:

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

CVTPD2PS—Convert Packed Double Precision Floating-Point Values to Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5A /r CVTPD2PS xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double precision floating-point values in xmm2/mem to two single precision floating-point values in xmm1.
VEX.128.66.0F.WIG 5A /r VCVTPD2PS xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double precision floating-point values in xmm2/mem to two single precision floating-point values in xmm1.
VEX.256.66.0F.WIG 5A /r VCVTPD2PS xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double precision floating-point values in ymm2/mem to four single precision floating-point values in xmm1.
EVEX.128.66.0F.W1 5A /r VCVTPD2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two single precision floating-point values in xmm1with writemask k1.
EVEX.256.66.0F.W1 5A /r VCVTPD2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four single precision floating-point values in xmm1with writemask k1.
EVEX.512.66.0F.W1 5A /r VCVTPD2PS ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight single precision floating-point values in ymm1with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Converts two, four or eight packed double precision floating-point values in the source operand (second operand) to two, four or eight packed single precision floating-point values in the destination operand (first operand).

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

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector 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-1:256/128/64) of the corresponding destination are zeroed.

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

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

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

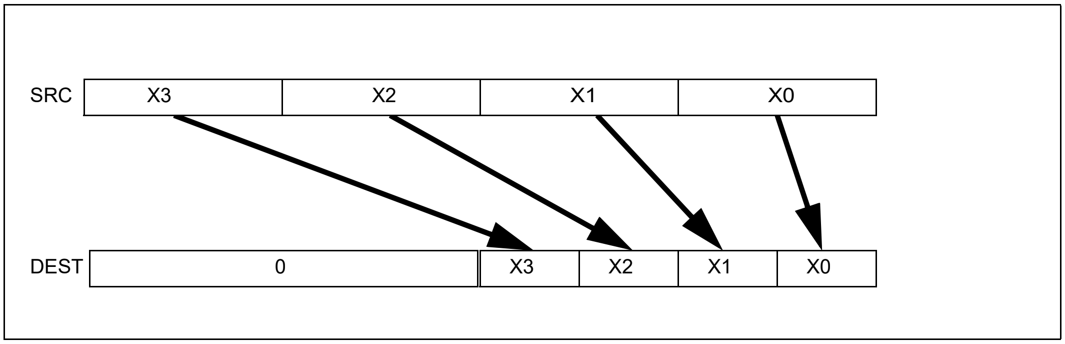


Figure 1-3. VCVTPD2PS (VEX.256 Encoded Version)

Operation

VCVTPD2PS (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] := Convert_Double_Precision_Floating_Point_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
```

VCVTPD2PS (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_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+31:i] := Convert_Double_Precision_Floating_Point_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
```

VCVTPD2PS (VEX.256 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[95:64] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[191:128])
DEST[127:96] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[255:192)
DEST[MAXVL-1:128] := 0
```

VCVTPD2PS (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[MAXVL-1:64] := 0
```

CVTPD2PS (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[127:64] := 0
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2PS __m256 _mm512_cvtpd_ps(__m512d a);
VCVTPD2PS __m256 _mm512_mask_cvtpd_ps(__m256 s, __mmask8 k, __m512d a);
VCVTPD2PS __m256 _mm512_maskz_cvtpd_ps(__mmask8 k, __m512d a);
VCVTPD2PS __m256 _mm512_cvt_roundpd_ps(__m512d a, int r);
VCVTPD2PS __m256 _mm512_mask_cvt_roundpd_ps(__m256 s, __mmask8 k, __m512d a, int r);
VCVTPD2PS __m256 _mm512_maskz_cvt_roundpd_ps(__mmask8 k, __m512d a, int r);
VCVTPD2PS __m128 _mm256_mask_cvtpd_ps(__m128 s, __mmask8 k, __m256d a);
VCVTPD2PS __m128 _mm256_maskz_cvtpd_ps(__mmask8 k, __m256d a);
VCVTPD2PS __m128 _mm_mask_cvtpd_ps(__m128 s, __mmask8 k, __m128d a);
VCVTPD2PS __m128 _mm_maskz_cvtpd_ps(__mmask8 k, __m128d a);
VCVTPD2PS __m128 _mm256_cvtpd_ps (__m256d a)
CVTPD2PS __m128 _mm_cvtpd_ps (__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision, Underflow, Overflow, Denormal.

Other Exceptions

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

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

Additionally:

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

CVTPS2DQ—Convert Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5B /r CVTPS2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.
VEX.128.66.0F.WIG 5B /r VCVTPS2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.
VEX.256.66.0F.WIG 5B /r VCVTPS2DQ ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed single precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1.
EVEX.128.66.0F.W0 5B /r VCVTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 subject to writemask k1.
EVEX.256.66.0F.W0 5B /r VCVTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 5B /r VCVTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Converts four, eight or sixteen packed single precision floating-point values in the source operand to four, eight or sixteen signed 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 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 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.

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

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

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

Operation

VCVTPS2DQ (Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer(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
```

VCVTPS2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

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

FOR j := 0 TO 15
    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_Integer(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer(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
```

VCVTPS2DQ (VEX.256 Encoded Version)

```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96)
DEST[159:128] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[159:128])
DEST[191:160] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[191:160])
DEST[223:192] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[223:192])
DEST[255:224] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[255:224])
```

VCVTPS2DQ (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[MAXVL-1:128] := 0
```

CVTPS2DQ (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPS2DQ __m512i _mm512_cvtps_epi32( __m512 a);
VCVTPS2DQ __m512i _mm512_mask_cvtps_epi32( __m512i s, __mmask16 k, __m512 a);
VCVTPS2DQ __m512i _mm512_maskz_cvtps_epi32( __mmask16 k, __m512 a);
VCVTPS2DQ __m512i _mm512_cvt_roundps_epi32( __m512 a, int r);
VCVTPS2DQ __m512i _mm512_mask_cvt_roundps_epi32( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2DQ __m512i _mm512_maskz_cvt_roundps_epi32( __mmask16 k, __m512 a, int r);
VCVTPS2DQ __m256i _mm256_mask_cvtps_epi32( __m256i s, __mmask8 k, __m256 a);
VCVTPS2DQ __m256i _mm256_maskz_cvtps_epi32( __mmask8 k, __m256 a);
VCVTPS2DQ __m128i _mm_mask_cvtps_epi32( __m128i s, __mmask8 k, __m128 a);
VCVTPS2DQ __m128i _mm_maskz_cvtps_epi32( __mmask8 k, __m128 a);
VCVTPS2DQ __ m256i _mm256_cvtps_epi32 (__m256 a)
CVTPS2DQ __m128i _mm_cvtps_epi32 (__m128 a)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

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

Additionally:

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

CVTPS2PD—Convert Packed Single Precision Floating-Point Values to Packed Double Precision Floating-Point Values

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 5A /r CVTPS2PD xmm1, xmm2/m64	A	V/V	SSE2	Convert two packed single precision floating-point values in xmm2/m64 to two packed double precision floating-point values in xmm1.
VEX.128.0F.WIG 5A /r VCVTPS2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed single precision floating-point values in xmm2/m64 to two packed double precision floating-point values in xmm1.
VEX.256.0F.WIG 5A /r VCVTPS2PD ymm1, xmm2/m128	A	V/V	AVX	Convert four packed single precision floating-point values in xmm2/m128 to four packed double precision floating-point values in ymm1.
EVEX.128.0F.W0 5A /r VCVTPS2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert two packed single precision floating-point values in xmm2/m64/m32bcst to packed double precision floating-point values in xmm1 with writemask k1.
EVEX.256.0F.W0 5A /r VCVTPS2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert four packed single precision floating-point values in xmm2/m128/m32bcst to packed double precision floating-point values in ymm1 with writemask k1.
EVEX.512.0F.W0 5A /r VCVTPS2PD zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}	B	V/V	AVX512F OR AVX10.1[1]	Convert eight packed single precision floating-point values in ymm2/m256/b32bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Converts two, four or eight packed single precision floating-point values in the source operand (second operand) to two, four or eight packed double precision floating-point values in the destination operand (first operand).

EVEX encoded versions: 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.

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

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

128-bit Legacy SSE version: The source operand is an XMM register or 64- bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

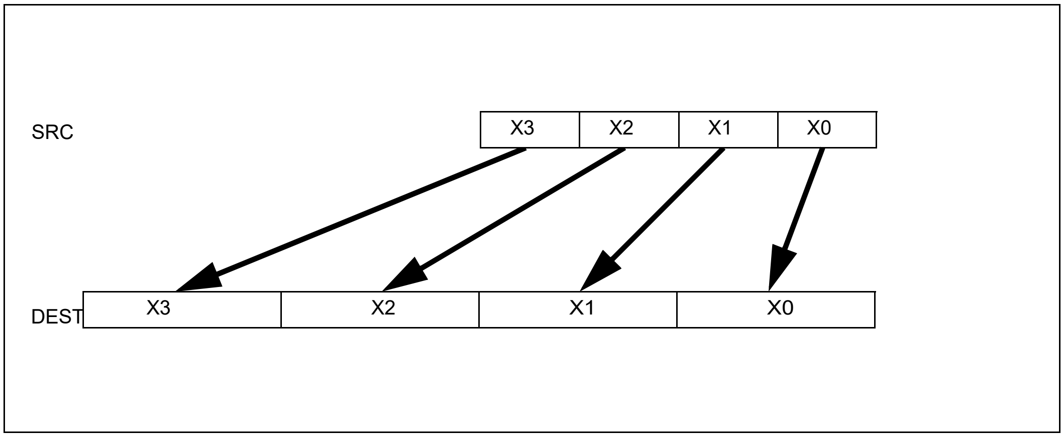


Figure 1-4. CVTPS2PD (VEX.256 Encoded Version)

Operation

VCVTPS2PD (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_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
```

VCVTPS2PD (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_Double_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Single_Precision_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
```

VCVTPS2PD (VEX.256 Encoded Version)

```
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[127:96)
DEST[MAXVL-1:256] := 0
```

VCVTPS2PD (VEX.128 Encoded Version)

```
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] := 0
```

CVTPS2PD (128-bit Legacy SSE Version)

```
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPS2PD __m512d _mm512_cvtps_pd( __m256 a);
VCVTPS2PD __m512d _mm512_mask_cvtps_pd( __m512d s, __mmask8 k, __m256 a);
VCVTPS2PD __m512d _mm512_maskz_cvtps_pd( __mmask8 k, __m256 a);
VCVTPS2PD __m512d _mm512_cvt_roundps_pd( __m256 a, int sae);
VCVTPS2PD __m512d _mm512_mask_cvt_roundps_pd( __m512d s, __mmask8 k, __m256 a, int sae);
VCVTPS2PD __m512d _mm512_maskz_cvt_roundps_pd( __mmask8 k, __m256 a, int sae);
VCVTPS2PD __m256d _mm256_mask_cvtps_pd( __m256d s, __mmask8 k, __m128 a);
VCVTPS2PD __m256d _mm256_maskz_cvtps_pd( __mmask8 k, __m128a);
VCVTPS2PD __m128d _mm_mask_cvtps_pd( __m128d s, __mmask8 k, __m128 a);
VCVTPS2PD __m128d _mm_maskz_cvtps_pd( __mmask8 k, __m128 a);
VCVTPS2PD __m256d _mm256_cvtps_pd (__m128 a)
CVTPS2PD __m128d _mm_cvtps_pd (__m128 a)
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

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

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

Additionally:

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

CVTSD2SI—Convert Scalar Double Precision Floating-Point Value to Doubleword Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2D /r CVTSD2SI r32, xmm1/m64	A	V/V	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
F2 REX.W 0F 2D /r CVTSD2SI r64, xmm1/m64	A	V/N.E.	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
VEX.LIG.F2.0F.W0 2D /r [1] VCVTSD2SI r32, xmm1/m64	A	V/V	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
VEX.LIG.F2.0F.W1 2D /r [1] VCVTSD2SI r64, xmm1/m64	A	V/N.E.[2]	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
EVEX.LLIG.F2.0F.W0 2D /r VCVTSD2SI r32, xmm1/m64{er}	B	V/V	AVX512F OR AVX10.1[3]	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
EVEX.LLIG.F2.0F.W1 2D /r VCVTSD2SI r64, xmm1/m64{er}	B	V/N.E.[2]	AVX512F OR AVX10.1[3]	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.

NOTES:

1. Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.
3. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Converts a double precision floating-point value in the source operand (the second operand) to a signed doubleword integer in the destination operand (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 exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000_00000000H) is returned.

Legacy SSE instruction: Use of the REX.W prefix promotes the instruction to produce 64-bit data in 64-bit mode. See the summary chart at the beginning of this section for encoding data and limits.

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

Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSD2SI (EVEX Encoded Version)

```
IF SRC *is register* AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode and OperandSize = 64
    THEN    DEST[63:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
    ELSE    DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
FI
```

(V)CVTSD2SI

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSD2SI int _mm_cvtsd_i32(__m128d);
VCVTSD2SI int _mm_cvt_roundsd_i32(__m128d, int r);
VCVTSD2SI __int64 _mm_cvtsd_i64(__m128d);
VCVTSD2SI __int64 _mm_cvt_roundsd_i64(__m128d, int r);
CVTSD2SI __int64 _mm_cvtsd_si64(__m128d);
CVTSD2SI int _mm_cvtsd_si32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

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

Additionally:

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

CVTSD2SS—Convert Scalar Double Precision Floating-Point Value to Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5A /r CVTSD2SS xmm1, xmm2/m64	A	V/V	SSE2	Convert one double precision floating-point value in xmm2/m64 to one single precision floating-point value in xmm1.
VEX.LIG.F2.0F.WIG 5A /r VCVTSD2SS xmm1,xmm2, xmm3/m64	B	V/V	AVX	Convert one double precision floating-point value in xmm3/m64 to one single precision floating-point value and merge with high bits in xmm2.
EVEX.LLIG.F2.0F.W1 5A /r VCVTSD2SS xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F OR AVX10.1[1]	Convert one double precision floating-point value in xmm3/m64 to one single precision floating-point value and merge with high bits in xmm2 under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Converts a double precision floating-point value in the "convert-from" source operand (the second operand in SSE2 version, otherwise the third operand) to a single precision floating-point value in the destination operand.

When the "convert-from" operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

128-bit Legacy SSE version: The "convert-from" source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result in written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSD2SS is encoded with VEX.L=0. Encoding VCVTSD2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSD2SS (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
    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
```

VCVTSD2SS (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

CVTSD2SS (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[MAXVL-1:32] Unmodified *)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSD2SS __m128 _mm_mask_cvtsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128 _mm_maskz_cvtsd_ss( __mmask8 k, __m128 a,__m128d b);
VCVTSD2SS __m128 _mm_cvt_roundsd_ss(__m128 a, __m128d b, int r);
VCVTSD2SS __m128 _mm_mask_cvt_roundsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b, int r);
VCVTSD2SS __m128 _mm_maskz_cvt_roundsd_ss( __mmask8 k, __m128 a,__m128d b, int r);
CVTSD2SS __m128_mm_cvtsd_ss(__m128 a, __m128d b)
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

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

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

CVTSI2SD—Convert Doubleword Integer to Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2A /r CVTSI2SD xmm1, r32/m32	A	V/V	SSE2	Convert one signed doubleword integer from r32/m32 to one double precision floating-point value in xmm1.
F2 REX.W 0F 2A /r CVTSI2SD xmm1, r/m64	A	V/N.E.	SSE2	Convert one signed quadword integer from r/m64 to one double precision floating-point value in xmm1.
VEX.LIG.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	B	V/V	AVX	Convert one signed doubleword integer from r/m32 to one double precision floating-point value in xmm1.
VEX.LIG.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64	B	V/N.E.[1]	AVX	Convert one signed quadword integer from r/m64 to one double precision floating-point value in xmm1.
EVEX.LLIG.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	C	V/V	AVX512F OR AVX10.1[1]	Convert one signed doubleword integer from r/m32 to one double precision floating-point value in xmm1.
EVEX.LLIG.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64{er}	C	V/N.E.[2]	AVX512F OR AVX10.1[1]	Convert one signed quadword integer from r/m64 to one double precision floating-point value in xmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.
2. VEX.W1/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	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the “convert-from” 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, and the high quadword left unchanged. 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.

128-bit Legacy SSE version: Use of the REX.W prefix promotes the instruction to 64-bit operands. The “convert-from” source operand (the second operand) is a general-purpose register or memory location. The destination is an XMM register Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

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

Software should ensure VCVTSI2SD is encoded with VEX.L=0. Encoding VCVTSI2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSI2SD (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VCVTSI2SD (VEX.128 Encoded Version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

CVTSI2SD

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSI2SD __m128d _mm_cvti32_sd(__m128d s, int a);
VCVTSI2SD __m128d _mm_cvti64_sd(__m128d s, __int64 a);
VCVTSI2SD __m128d _mm_cvt_roundi64_sd(__m128d s, __int64 a, int r);
CVTSI2SD __m128d _mm_cvtsi64_sd(__m128d s, __int64 a);
CVTSI2SD __m128d_mm_cvtsi32_sd(__m128d a, int b)
```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions," if W1; else see Table 2-22, "Type 5 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions,” if W1; else see Table 2-59, “Type E10NF Class Exception Conditions.”

CVTSI2SS—Convert Doubleword Integer to Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2A /r CVTSI2SS xmm1, r/m32	A	V/V	SSE	Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.
F3 REX.W 0F 2A /r CVTSI2SS xmm1, r/m64	A	V/N.E.	SSE	Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.
VEX.LIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32	B	V/V	AVX	Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.
VEX.LIG.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64	B	V/N.E.[1]	AVX	Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.
EVEX.LLIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32{er}	C	V/V	AVX512F OR AVX10.1[1]	Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.
EVEX.LLIG.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64{er}	C	V/N.E.[2]	AVX512F OR AVX10.1[1]	Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.
2. VEX.W1/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	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the “convert-from” source operand to a single precision floating-point value in the destination operand (first operand). The “convert-from” 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, and the upper three doublewords are left unchanged. 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.

128-bit Legacy SSE version: In 64-bit mode, Use of the REX.W prefix promotes the instruction to use 64-bit input value. The “convert-from” source operand (the second operand) is a general-purpose register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be a general-purpose register or a 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 encoded version: the converted result in written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSI2SS is encoded with VEX.L=0. Encoding VCVTSI2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSI2SS (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VCVTSI2SS (VEX.128 Encoded Version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

CVTSI2SS (128-bit Legacy SSE Version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] :=Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSI2SS __m128 _mm_cvti32_ss(__m128 s, int a);
VCVTSI2SS __m128 _mm_cvt_roundi32_ss(__m128 s, int a, int r);
VCVTSI2SS __m128 _mm_cvti64_ss(__m128 s, __int64 a);
VCVTSI2SS __m128 _mm_cvt_roundi64_ss(__m128 s, __int64 a, int r);
CVTSI2SS __m128 _mm_cvtsi64_ss(__m128 s, __int64 a);
CVTSI2SS __m128 _mm_cvtsi32_ss(__m128 a, int b);
```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

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

CVTSS2SD—Convert Scalar Single Precision Floating-Point Value to Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5A /r CVTSS2SD xmm1, xmm2/m32	A	V/V	SSE2	Convert one single precision floating-point value in xmm2/m32 to one double precision floating-point value in xmm1.
VEX.LIG.F3.0F.WIG 5A /r VCVTSS2SD xmm1, xmm2, xmm3/m32	B	V/V	AVX	Convert one single precision floating-point value in xmm3/m32 to one double precision floating-point value and merge with high bits of xmm2.
EVEX.LLIG.F3.0F.W0 5A /r VCVTSS2SD xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F OR AVX10.1[1]	Convert one single precision floating-point value in xmm3/m32 to one double precision floating-point value and merge with high bits of xmm2 under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Converts a single precision floating-point value in the “convert-from” source operand to a double precision floating-point value in the destination operand. When the “convert-from” source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand.

128-bit Legacy SSE version: The “convert-from” source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

Software should ensure VCVTSS2SD is encoded with VEX.L=0. Encoding VCVTSS2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSS2SD (EVEX Encoded Version)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0]);
    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
```

VCVTSS2SD (VEX.128 Encoded Version)

```
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0])
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

CVTSS2SD (128-bit Legacy SSE Version)

```
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSS2SD __m128d _mm_cvt_roundss_sd(__m128d a, __m128 b, int r);
VCVTSS2SD __m128d _mm_mask_cvt_roundss_sd(__m128d s, __mmask8 m, __m128d a,__m128 b, int r);
VCVTSS2SD __m128d _mm_maskz_cvt_roundss_sd(__mmask8 k, __m128d a, __m128 a, int r);
VCVTSS2SD __m128d _mm_mask_cvtss_sd(__m128d s, __mmask8 m, __m128d a,__m128 b);
VCVTSS2SD __m128d _mm_maskz_cvtss_sd(__mmask8 m, __m128d a,__m128 b);
CVTSS2SD __m128d_mm_cvtss_sd(__m128d a, __m128 a);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

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

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

CVTSS2SI—Convert Scalar Single Precision Floating-Point Value to Doubleword Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2D /r CVTSS2SI r32, xmm1/m32	A	V/V	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
F3 REX.W 0F 2D /r CVTSS2SI r64, xmm1/m32	A	V/N.E.	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
VEX.LIG.F3.0F.W0 2D /r [1] VCVTSS2SI r32, xmm1/m32	A	V/V	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
VEX.LIG.F3.0F.W1 2D /r [1] VCVTSS2SI r64, xmm1/m32	A	V/N.E.[2]	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
EVEX.LLIG.F3.0F.W0 2D /r VCVTSS2SI r32, xmm1/m32{er}	B	V/V	AVX512F OR AVX10.1[3]	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
EVEX.LLIG.F3.0F.W1 2D /r VCVTSS2SI r64, xmm1/m32{er}	B	V/N.E.[2]	AVX512F OR AVX10.1[3]	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64.

NOTES:

1. Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.
3. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Converts a single precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed 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 indefinite integer value (2^w-1, where w represents the number of bits in the destination format) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to produce 64-bit data. See the summary chart at the beginning of this section for encoding data and limits.

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

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

Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSS2SI (EVEX Encoded Version)

```
IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
ELSE
    DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
FI;
```

(V)CVTSS2SI (Legacy and VEX.128 Encoded Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2SI int _mm_cvtss_i32(__m128 a);
VCVTSS2SI int _mm_cvt_roundss_i32(__m128 a, int r);
VCVTSS2SI __int64 _mm_cvtss_i64(__m128 a);
VCVTSS2SI __int64 _mm_cvt_roundss_i64(__m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions,” additionally:

#UD	If VEX.vvvv != 1111B.

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

CVTTPD2DQ—Convert with Truncation Packed Double Precision Floating-Point Values to Packed Doubleword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E6 /r CVTTPD2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.
VEX.128.66.0F.WIG E6 /r VCVTTPD2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.
VEX.256.66.0F.WIG E6 /r VCVTTPD2DQ xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1 using truncation.
EVEX.128.66.0F.W1 E6 /r VCVTTPD2DQ xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W1 E6 /r VCVTTPD2DQ xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W1 E6 /r VCVTTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst{sae}	B	V/V	AVX512F OR AVX10.1[1]	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 using truncation subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Converts two, four or eight packed double precision floating-point values in the source operand (second operand) to two, four or eight packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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 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.

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

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

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

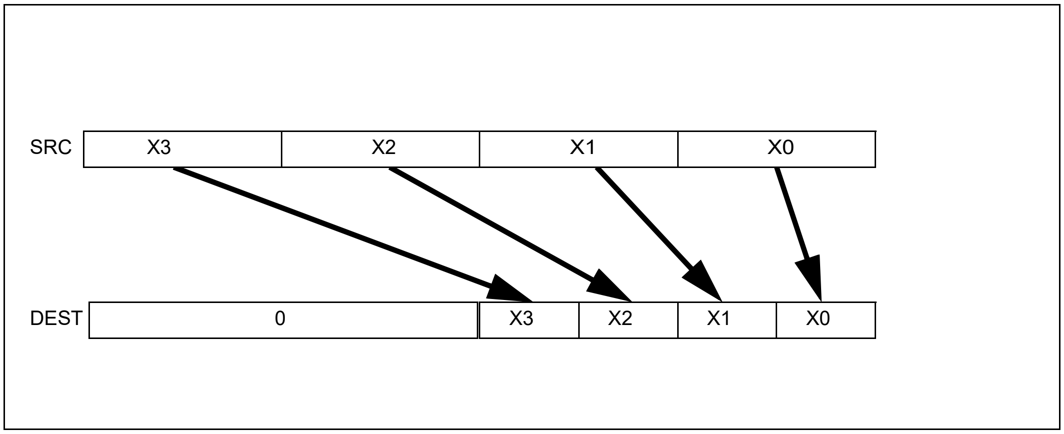


Figure 1-5. VCVTTPD2DQ (VEX.256 Encoded Version)

Operation

VCVTTPD2DQ (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 * 32
    k := j * 64
    IF k1[j] OR *no writemask*
         THEN DEST[i+31:i] :=
              Convert_Double_Precision_Floating_Point_To_Integer_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
```

VCVTTPD2DQ (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_Integer_Truncate(SRC[63:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer_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
```

VCVTTPD2DQ (VEX.256 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[95:64] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[191:128])
DEST[127:96] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[255:192])
DEST[MAXVL-1:128] := 0
```

VCVTTPD2DQ (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[MAXVL-1:64] := 0
```

CVTTPD2DQ (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[127:64] := 0
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2DQ __m256i _mm512_cvttpd_epi32(__m512d a);
VCVTTPD2DQ __m256i _mm512_mask_cvttpd_epi32(__m256i s, __mmask8 k, __m512d a);
VCVTTPD2DQ __m256i _mm512_maskz_cvttpd_epi32(__mmask8 k, __m512d a);
VCVTTPD2DQ __m256i _mm512_cvtt_roundpd_epi32(__m512d a, int sae);
VCVTTPD2DQ __m256i _mm512_mask_cvtt_roundpd_epi32(__m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2DQ __m256i _mm512_maskz_cvtt_roundpd_epi32(__mmask8 k, __m512d a, int sae);
VCVTTPD2DQ __m128i _mm256_mask_cvttpd_epi32(__m128i s, __mmask8 k, __m256d a);
VCVTTPD2DQ __m128i _mm256_maskz_cvttpd_epi32(__mmask8 k, __m256d a);
VCVTTPD2DQ __m128i _mm_mask_cvttpd_epi32(__m128i s, __mmask8 k, __m128d a);
VCVTTPD2DQ __m128i _mm_maskz_cvttpd_epi32(__mmask8 k, __m128d a);
VCVTTPD2DQ __m128i _mm256_cvttpd_epi32 (__m256d src);
CVTTPD2DQ __m128i _mm_cvttpd_epi32 (__m128d src);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

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

Additionally:

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

CVTTPS2DQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5B /r CVTTPS2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.128.F3.0F.WIG 5B /r VCVTTPS2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.256.F3.0F.WIG 5B /r VCVTTPS2DQ ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed single precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1 using truncation.
EVEX.128.F3.0F.W0 5B /r VCVTTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 using truncation subject to writemask k1.
EVEX.256.F3.0F.W0 5B /r VCVTTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 using truncation subject to writemask k1.
EVEX.512.F3.0F.W0 5B /r VCVTTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	B	V/V	AVX512F OR AVX10.1[1]	Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 using truncation subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Converts four, eight or sixteen packed single precision floating-point values in the source operand to four, eight or sixteen signed doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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.

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

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

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

Operation

VCVTTPS2DQ (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_Integer_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
```

VCVTTPS2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

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

FOR j := 0 TO 15
    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_Integer_Truncate(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer_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
```

VCVTTPS2DQ (VEX.256 Encoded Version)
```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96)
DEST[159:128] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[159:128])
DEST[191:160] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[191:160])
DEST[223:192] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[223:192])
DEST[255:224] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[255:224])
```

VCVTTPS2DQ (VEX.128 encoded version)
```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[MAXVL-1:128] := 0
```

CVTTPS2DQ (128-bit Legacy SSE version)
```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTPS2DQ __m512i _mm512_cvttps_epi32( __m512 a);
VCVTTPS2DQ __m512i _mm512_mask_cvttps_epi32( __m512i s, __mmask16 k, __m512 a);
VCVTTPS2DQ __m512i _mm512_maskz_cvttps_epi32( __mmask16 k, __m512 a);
VCVTTPS2DQ __m512i _mm512_cvtt_roundps_epi32( __m512 a, int sae);
VCVTTPS2DQ __m512i _mm512_mask_cvtt_roundps_epi32( __m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2DQ __m512i _mm512_maskz_cvtt_roundps_epi32( __mmask16 k, __m512 a, int sae);
VCVTTPS2DQ __m256i _mm256_mask_cvttps_epi32( __m256i s, __mmask8 k, __m256 a);
VCVTTPS2DQ __m256i _mm256_maskz_cvttps_epi32( __mmask8 k, __m256 a);
VCVTTPS2DQ __m128i _mm_mask_cvttps_epi32( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2DQ __m128i _mm_maskz_cvttps_epi32( __mmask8 k, __m128 a);
VCVTTPS2DQ __m256i _mm256_cvttps_epi32 (__m256 a)
CVTTPS2DQ __m128i _mm_cvttps_epi32 (__m128 a)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

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

Additionally:

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

CVTTSD2SI—Convert With Truncation Scalar Double Precision Floating-Point Value to Signed Integer

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
F2 0F 2C /r CVTTSD2SI r32, xmm1/m64	A	V/V	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
F2 REX.W 0F 2C /r CVTTSD2SI r64, xmm1/m64	A	V/N.E.	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
VEX.LIG.F2.0F.W0 2C /r [1] VCVTTSD2SI r32, xmm1/m64	A	V/V	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
VEX.LIG.F2.0F.W1 2C /r [1] VCVTTSD2SI r64, xmm1/m64	B	V/N.E.[2]	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
EVEX.LLIG.F2.0F.W0 2C /r VCVTTSD2SI r32, xmm1/m64{sae}	B	V/V	AVX512F OR AVX10.1[3]	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
EVEX.LLIG.F2.0F.W1 2C /r VCVTTSD2SI r64, xmm1/m64{sae}	B	V/N.E.[2]	AVX512F OR AVX10.1[3]	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.

NOTES:

1. Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.
3. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Converts a double precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed 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 exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000_00000000H) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

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

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

Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)CVTTSD2SI (All Versions)

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSD2SI int _mm_cvttsd_i32(__m128d a);
VCVTTSD2SI int _mm_cvtt_roundsd_i32(__m128d a, int sae);
VCVTTSD2SI __int64 _mm_cvttsd_i64(__m128d a);
VCVTTSD2SI __int64 _mm_cvtt_roundsd_i64(__m128d a, int sae);
CVTTSD2SI int _mm_cvttsd_si32(__m128d a);
CVTTSD2SI __int64 _mm_cvttsd_si64(__m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions,” additionally:

#UD	If VEX.vvvv != 1111B.

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

CVTTSS2SI—Convert With Truncation Scalar Single Precision Floating-Point Value to Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2C /r CVTTSS2SI r32, xmm1/m32	A	V/V	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
F3 REX.W 0F 2C /r CVTTSS2SI r64, xmm1/m32	A	V/N.E.	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
VEX.LIG.F3.0F.W0 2C /r [1] VCVTTSS2SI r32, xmm1/m32	A	V/V	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
VEX.LIG.F3.0F.W1 2C /r [1] VCVTTSS2SI r64, xmm1/m32	A	V/N.E.[2]	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
EVEX.LLIG.F3.0F.W0 2C /r VCVTTSS2SI r32, xmm1/m32{sae}	B	V/V	AVX512F OR AVX10.1[3]	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
EVEX.LLIG.F3.0F.W1 2C /r VCVTTSS2SI r64, xmm1/m32{sae}	B	V/N.E.[2]	AVX512F OR AVX10.1[3]	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.

NOTES:

1. Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.
3. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Converts a single precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed 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 32-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H or 80000000_00000000H if operand size is 64 bits) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

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

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

Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)CVTTSS2SI (All Versions)

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSS2SI int _mm_cvttss_i32(__m128 a);
VCVTTSS2SI int _mm_cvtt_roundss_i32(__m128 a, int sae);
VCVTTSS2SI __int64 _mm_cvttss_i64(__m128 a);
VCVTTSS2SI __int64 _mm_cvtt_roundss_i64(__m128 a, int sae);
CVTTSS2SI int _mm_cvttss_si32(__m128 a);
CVTTSS2SI __int64 _mm_cvttss_si64(__m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 2-20, “Type 3 Class Exception Conditions,” additionally:

#UD	If VEX.vvvv != 1111B.

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

DIVPD—Divide Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5E /r DIVPD xmm1, xmm2/m128	A	V/V	SSE2	Divide packed double precision floating-point values in xmm1 by packed double precision floating-point values in xmm2/mem.
VEX.128.66.0F.WIG 5E /r VDIVPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Divide packed double precision floating-point values in xmm2 by packed double precision floating-point values in xmm3/mem.
VEX.256.66.0F.WIG 5E /r VDIVPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Divide packed double precision floating-point values in ymm2 by packed double precision floating-point values in ymm3/mem.
EVEX.128.66.0F.W1 5E /r VDIVPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Divide packed double precision floating-point values in xmm2 by packed double precision floating-point values in xmm3/m128/m64bcst and write results to xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 5E /r VDIVPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Divide packed double precision floating-point values in ymm2 by packed double precision floating-point values in ymm3/m256/m64bcst and write results to ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 5E /r VDIVPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F OR AVX10.1[1]	Divide packed double precision floating-point values in zmm2 by packed double precision floating-point values in zmm3/m512/m64bcst and write results to zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a SIMD divide of the double precision floating-point values in the first source operand by the floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

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

VEX.256 encoded version: The first source operand (the second operand) is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

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

128-bit Legacy SSE version: The second source operand (the second operand) can be an XMM register or an 128-bit memory location. The destination is the same as the first source operand. The upper bits (MAXVL-1:128) of the corresponding destination are unmodified.

Operation

VDIVPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);  ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures
Software Developer's Manual, Volume 1
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := SRC1[i+63:i] / SRC2[63:0]
                ELSE
                    DEST[i+63:i] := 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
```

VDIVPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] / SRC2[63:0]
DEST[127:64] := SRC1[127:64] / SRC2[127:64]
DEST[191:128] := SRC1[191:128] / SRC2[191:128]
DEST[255:192] := SRC1[255:192] / SRC2[255:192]
DEST[MAXVL-1:256] := 0;
```

VDIVPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] / SRC2[63:0]
DEST[127:64] := SRC1[127:64] / SRC2[127:64]
DEST[MAXVL-1:128] := 0;
```

DIVPD (128-bit Legacy SSE Version)

```
DEST[63:0] := SRC1[63:0] / SRC2[63:0]
DEST[127:64] := SRC1[127:64] / SRC2[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VDIVPD __m512d _mm512_div_pd(__m512d a, __m512d b);
VDIVPD __m512d _mm512_mask_div_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VDIVPD __m512d _mm512_maskz_div_pd(__mmask8 k, __m512d a, __m512d b);
VDIVPD __m256d _mm256_mask_div_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d _mm256_maskz_div_pd(__mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d _mm_mask_div_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d _mm_maskz_div_pd(__mmask8 k, __m128d a, __m128d b);
VDIVPD __m512d _mm512_div_round_pd(__m512d a, __m512d b, int);
VDIVPD __m512d _mm512_mask_div_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m512d _mm512_maskz_div_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m256d _mm256_div_pd (__m256d a, __m256d b);
DIVPD __m128d _mm_div_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

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

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

DIVPS—Divide Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5E /r DIVPS xmm1, xmm2/m128	A	V/V	SSE	Divide packed single precision floating-point values in xmm1 by packed single precision floating-point values in xmm2/mem.
VEX.128.0F.WIG 5E /r VDIVPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Divide packed single precision floating-point values in xmm2 by packed single precision floating-point values in xmm3/mem.
VEX.256.0F.WIG 5E /r VDIVPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Divide packed single precision floating-point values in ymm2 by packed single precision floating-point values in ymm3/mem.
EVEX.128.0F.W0 5E /r VDIVPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Divide packed single precision floating-point values in xmm2 by packed single precision floating-point values in xmm3/m128/m32bcst and write results to xmm1 subject to writemask k1.
EVEX.256.0F.W0 5E /r VDIVPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Divide packed single precision floating-point values in ymm2 by packed single precision floating-point values in ymm3/m256/m32bcst and write results to ymm1 subject to writemask k1.
EVEX.512.0F.W0 5E /r VDIVPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	C	V/V	AVX512F OR AVX10.1[1]	Divide packed single precision floating-point values in zmm2 by packed single precision floating-point values in zmm3/m512/m32bcst and write results to zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a SIMD divide of the four, eight or sixteen packed single precision floating-point values in the first source operand (the second operand) by the four, eight or sixteen packed single precision floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

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

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. 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 ZMM register destination are unmodified.

Operation

VDIVPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
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] / SRC2[31:0]
                ELSE
                    DEST[i+31:i] := 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
```

VDIVPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] / SRC2[31:0]
DEST[63:32] := SRC1[63:32] / SRC2[63:32]
DEST[95:64] := SRC1[95:64] / SRC2[95:64]
DEST[127:96] := SRC1[127:96] / SRC2[127:96]
DEST[159:128] := SRC1[159:128] / SRC2[159:128]
DEST[191:160] := SRC1[191:160] / SRC2[191:160]
DEST[223:192] := SRC1[223:192] / SRC2[223:192]
DEST[255:224] := SRC1[255:224] / SRC2[255:224].
DEST[MAXVL-1:256] := 0;
```

VDIVPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] / SRC2[31:0]
DEST[63:32] := SRC1[63:32] / SRC2[63:32]
DEST[95:64] := SRC1[95:64] / SRC2[95:64]
DEST[127:96] := SRC1[127:96] / SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

DIVPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0] / SRC2[31:0]
DEST[63:32] := SRC1[63:32] / SRC2[63:32]
DEST[95:64] := SRC1[95:64] / SRC2[95:64]
DEST[127:96] := SRC1[127:96] / SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VDIVPS __m512 _mm512_div_ps( __m512 a, __m512 b);
VDIVPS __m512 _mm512_mask_div_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VDIVPS __m512 _mm512_maskz_div_ps(__mmask16 k, __m512 a, __m512 b);
VDIVPD __m256d _mm256_mask_div_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d _mm256_maskz_div_pd( __mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d _mm_mask_div_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d _mm_maskz_div_pd( __mmask8 k, __m128d a, __m128d b);
VDIVPS __m512 _mm512_div_round_ps( __m512 a, __m512 b, int);
VDIVPS __m512 _mm512_mask_div_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VDIVPS __m512 _mm512_maskz_div_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VDIVPS __m256 _mm256_div_ps (__m256 a, __m256 b);
DIVPS __m128 _mm_div_ps (__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

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

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

DIVSD—Divide Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5E /r DIVSD xmm1, xmm2/m64	A	V/V	SSE2	Divide low double precision floating-point value in xmm1 by low double precision floating-point value in xmm2/m64.
VEX.LIG.F2.0F.WIG 5E /r VDIVSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Divide low double precision floating-point value in xmm2 by low double precision floating-point value in xmm3/m64.
EVEX.LLIG.F2.0F.W1 5E /r VDIVSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F OR AVX10.1[1]	Divide low double precision floating-point value in xmm2 by low double precision floating-point value in xmm3/m64.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Divides the low double precision floating-point value in the first source operand by the low double precision floating-point value in the second source operand, and stores the double precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The quadword at bits 127:64 of the destination operand is copied from the corresponding quadword of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The quadword element of the destination operand at bits 127:64 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

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

Software should ensure VDIVSD is encoded with VEX.L=0. Encoding VDIVSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VDIVSD (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC1[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] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VDIVSD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] / SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

DIVSD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] / SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VDIVSD __m128d _mm_mask_div_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_maskz_div_sd( __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_div_round_sd( __m128d a, __m128d b, int);
VDIVSD __m128d _mm_mask_div_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VDIVSD __m128d _mm_maskz_div_round_sd( __mmask8 k, __m128d a, __m128d b, int);
DIVSD __m128d _mm_div_sd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

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

DIVSS—Divide Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5E /r DIVSS xmm1, xmm2/m32	A	V/V	SSE	Divide low single precision floating-point value in xmm1 by low single precision floating-point value in xmm2/m32.
VEX.LIG.F3.0F.WIG 5E /r VDIVSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Divide low single precision floating-point value in xmm2 by low single precision floating-point value in xmm3/m32.
EVEX.LLIG.F3.0F.W0 5E /r VDIVSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F OR AVX10.1[1]	Divide low single precision floating-point value in xmm2 by low single precision floating-point value in xmm3/m32.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Divides the low single precision floating-point value in the first source operand by the low single precision floating-point value in the second source operand, and stores the single precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The doubleword elements of the destination operand at bits 127:32 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

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

Software should ensure VDIVSS is encoded with VEX.L=0. Encoding VDIVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VDIVSS (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC1[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] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VDIVSS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] / SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

DIVSS (128-bit Legacy SSE Version)

```
DEST[31:0] := DEST[31:0] / SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VDIVSS __m128 _mm_mask_div_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_maskz_div_ss( __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_div_round_ss( __m128 a, __m128 b, int);
VDIVSS __m128 _mm_mask_div_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VDIVSS __m128 _mm_maskz_div_round_ss( __mmask8 k, __m128 a, __m128 b, int);
DIVSS __m128 _mm_div_ss(__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

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

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

EXTRACTPS—Extract Packed Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 17 /r ib EXTRACTPS reg/m32, xmm1, imm8	A	VV	SSE4_1	Extract one single precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.
VEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8	A	V/V	AVX	Extract one single precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.
EVEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8	B	V/V	AVX512F OR AVX10.1[1]	Extract one single precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
B	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

Extracts a single precision floating-point value from the source operand (second operand) at the 32-bit offset specified from imm8. Immediate bits higher than the most significant offset for the vector length are ignored.

The extracted single precision floating-point value is stored in the low 32-bits of the destination operand

In 64-bit mode, destination register operand has default operand size of 64 bits. The upper 32-bits of the register are filled with zero. REX.W is ignored.

VEX.128 and EVEX encoded version: When VEX.W1 or EVEX.W1 form is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

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

128-bit Legacy SSE version: When a REX.W prefix is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

The source register is an XMM register. Imm8[1:0] determine the starting DWORD offset from which to extract the 32-bit floating-point value.

If VEXTRACTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

VEXTRACTPS (EVEX and VEX.128 Encoded Version)

```
SRC_OFFSET := IMM8[1:0]
IF (64-Bit Mode and DEST is register)
    DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
    DEST[63:32] := 0
ELSE
    DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
FI
```

EXTRACTPS (128-bit Legacy SSE Version)

```
SRC_OFFSET := IMM8[1:0]
IF (64-Bit Mode and DEST is register)
    DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
    DEST[63:32] := 0
ELSE
    DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

EXTRACTPS int _mm_extract_ps (__m128 a, const int nidx);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-57, “Type E9NF Class Exception Conditions.”

Additionally:

#UD	IF VEX.L = 0.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

GF2P8AFFINEINVQB—Galois Field Affine Transformation Inverse

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F3A CF /r /ib GF2P8AFFINEINVQB xmm1, xmm2/m128, imm8	A	V/V	GFNI	Computes inverse affine transformation in the finite field GF(2^8).
VEX.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX GFNI	Computes inverse affine transformation in the finite field GF(2^8).
VEX.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX GFNI	Computes inverse affine transformation in the finite field GF(2^8).
EVEX.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	(AVX512VL OR AVX10.1[1]) GFNI	Computes inverse affine transformation in the finite field GF(2^8).
EVEX.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	(AVX512VL OR AVX10.1[1]) GFNI	Computes inverse affine transformation in the finite field GF(2^8).
EVEX.512.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	(AVX512F OR AVX10.1[1]) GFNI	Computes inverse affine transformation in the finite field GF(2^8).

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8 (r)	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

The AFFINEINVB instruction computes an affine transformation in the Galois Field 2^8. For this instruction, an affine transformation is defined by A * inv(x) + b where "A" is an 8 by 8 bit matrix, and "x" and "b" are 8-bit vectors. The inverse of the bytes in x is defined with respect to the reduction polynomial $x^8 + x^4 + x^3 + x + 1$.

One SIMD register (operand 1) holds "x" as either 16, 32 or 64 8-bit vectors. A second SIMD (operand 2) register or memory operand contains 2, 4, or 8 "A" values, which are operated upon by the correspondingly aligned 8 "x" values in the first register. The "b" vector is constant for all calculations and contained in the immediate byte.

The EVEX encoded form of this instruction does not support memory fault suppression. The SSE encoded forms of the instruction require 16B alignment on their memory operations.

The inverse of each byte is given by the following table. The upper nibble is on the vertical axis and the lower nibble is on the horizontal axis. For example, the inverse of 0x95 is 0x8A.

Table 1-4. Inverse Byte Listings

-	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	8D	F6	CB	52	7B	D1	E8	4F	29	C0	B0	E1	E5	C7
1	74	B4	AA	4B	99	2B	60	5F	58	3F	FD	CC	FF	40	EE	B2
2	3A	6E	5A	F1	55	4D	A8	C9	C1	A	98	15	30	44	A2	C2
3	2C	45	92	6C	F3	39	66	42	F2	35	20	6F	77	BB	59	19
4	1D	FE	37	67	2D	31	F5	69	A7	64	AB	13	54	25	E9	9
5	ED	5C	5	CA	4C	24	87	BF	18	3E	22	F0	51	EC	61	17
6	16	5E	AF	D3	49	A6	36	43	F4	47	91	DF	33	93	21	3B
7	79	B7	97	85	10	B5	BA	3C	B6	70	D0	6	A1	FA	81	82
8	83	7E	7F	80	96	73	BE	56	9B	9E	95	D9	F7	2	B9	A4
9	DE	6A	32	6D	D8	8A	84	72	2A	14	9F	88	F9	DC	89	9A
A	FB	7C	2E	C3	8F	B8	65	48	26	C8	12	4A	CE	E7	D2	62
B	C	E0	1F	EF	11	75	78	71	A5	8E	76	3D	BD	BC	86	57
C	B	28	2F	A3	DA	D4	E4	F	A9	27	53	4	1B	FC	AC	E6
D	7A	7	AE	63	C5	DB	E2	EA	94	8B	C4	D5	9D	F8	90	6B
E	B1	D	D6	EB	C6	E	CF	AD	8	4E	D7	E3	5D	50	1E	B3
F	5B	23	38	34	68	46	3	8C	DD	9C	7D	A0	CD	1A	41	1C

Operation

```
define affine_inverse_byte(tsrc2qw, src1byte, imm):
    FOR i := 0 to 7:
        * parity(x) = 1 if x has an odd number of 1s in it, and 0 otherwise.*
        * inverse(x) is defined in the table above *
        retbyte.bit[i] := parity(tsrc2qw.byte[7-i] AND inverse(src1byte)) XOR imm8.bit[i]
    return retbyte
```

VGF2P8AFFINEINVQB dest, src1, src2, imm8 (EVEX Encoded Version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC2 is memory and EVEX.b==1:
        tsrc2 := SRC2.qword[0]
    ELSE:
        tsrc2 := SRC2.qword[j]

FOR b := 0 to 7:
    IF k1[j*8+b] OR *no writemask*:
        FOR i := 0 to 7:
            DEST.qword[j].byte[b] := affine_inverse_byte(tsrc2, SRC1.qword[j].byte[b], imm8)
    ELSE IF *zeroing*:
        DEST.qword[j].byte[b] := 0
    *ELSE DEST.qword[j].byte[b] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VGF2P8AFFINEINVQB dest, src1, src2, imm8 (128b and 256b VEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256)
FOR j := 0 TO KL-1:
    FOR b := 0 to 7:
        DEST.qword[j].byte[b] := affine_inverse_byte(SRC2.qword[j], SRC1.qword[j].byte[b], imm8)
DEST[MAX_VL-1:VL] := 0
```

GF2P8AFFINEINVQB srcdest, src1, imm8 (128b SSE Encoded Version)

```
FOR j := 0 TO 1:
    FOR b := 0 to 7:
        SRCDEST.qword[j].byte[b] := affine_inverse_byte(SRC1.qword[j], SRCDEST.qword[j].byte[b], imm8)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)GF2P8AFFINEINVQB __m128i _mm_gf2p8affineinv_epi64_epi8(__m128i, __m128i, int);
(V)GF2P8AFFINEINVQB __m128i _mm_mask_gf2p8affineinv_epi64_epi8(__m128i, __mmask16, __m128i, __m128i, int);
(V)GF2P8AFFINEINVQB __m128i _mm_maskz_gf2p8affineinv_epi64_epi8(__mmask16, __m128i, __m128i, int);
VGF2P8AFFINEINVQB __m256i _mm256_gf2p8affineinv_epi64_epi8(__m256i, __m256i, int);
VGF2P8AFFINEINVQB __m256i _mm256_mask_gf2p8affineinv_epi64_epi8(__m256i, __mmask32, __m256i, __m256i, int);
VGF2P8AFFINEINVQB __m256i _mm256_maskz_gf2p8affineinv_epi64_epi8(__mmask32, __m256i, __m256i, int);
VGF2P8AFFINEINVQB __m512i _mm512_gf2p8affineinv_epi64_epi8(__m512i, __m512i, int);
VGF2P8AFFINEINVQB __m512i _mm512_mask_gf2p8affineinv_epi64_epi8(__m512i, __mmask64, __m512i, __m512i, int);
VGF2P8AFFINEINVQB __m512i _mm512_maskz_gf2p8affineinv_epi64_epi8(__mmask64, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Legacy-encoded and VEX-encoded: See Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded: See Table 2-50, "Type E4NF Class Exception Conditions."

GF2P8AFFINEQB—Galois Field Affine Transformation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F3A CE /r /ib GF2P8AFFINEQB xmm1, xmm2/m128, imm8	A	V/V	GFNI	Computes affine transformation in the finite field GF(2^8).
VEX.128.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX GFNI	Computes affine transformation in the finite field GF(2^8).
VEX.256.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX GFNI	Computes affine transformation in the finite field GF(2^8).
EVEX.128.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	(AVX512VL OR AVX10.1[1]) GFNI	Computes affine transformation in the finite field GF(2^8).
EVEX.256.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	(AVX512VL OR AVX10.1[1]) GFNI	Computes affine transformation in the finite field GF(2^8).
EVEX.512.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	(AVX512F OR AVX10.1[1]) GFNI	Computes affine transformation in the finite field GF(2^8).

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8 (r)	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

The AFFINEB instruction computes an affine transformation in the Galois Field 2^8. For this instruction, an affine transformation is defined by A * x + b where “A” is an 8 by 8 bit matrix, and “x” and “b” are 8-bit vectors. One SIMD register (operand 1) holds “x” as either 16, 32 or 64 8-bit vectors. A second SIMD (operand 2) register or memory operand contains 2, 4, or 8 “A” values, which are operated upon by the correspondingly aligned 8 “x” values in the first register. The “b” vector is constant for all calculations and contained in the immediate byte.

The EVEX encoded form of this instruction does not support memory fault suppression. The SSE encoded forms of the instruction require16B alignment on their memory operations.

Operation

```
define parity(x):
    t := 0                    // single bit
    FOR i := 0 to 7:
        t = t xor x.bit[i]
    return t

define affine_byte(tsrc2qw, src1byte, imm):
    FOR i := 0 to 7:
        * parity(x) = 1 if x has an odd number of 1s in it, and 0 otherwise.*
        retbyte.bit[i] := parity(tsrc2qw.byte[7-i] AND src1byte) XOR imm8.bit[i]
    return retbyte
```

VGF2P8AFFINEQB dest, src1, src2, imm8 (EVEX Encoded Version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC2 is memory and EVEX.b==1:
        tsrc2 := SRC2.qword[0]
    ELSE:
        tsrc2 := SRC2.qword[j]

    FOR b := 0 to 7:
        IF k1[j*8+b] OR *no writemask*:
            DEST.qword[j].byte[b] := affine_byte(tsrc2, SRC1.qword[j].byte[b], imm8)
        ELSE IF *zeroing*:
            DEST.qword[j].byte[b] := 0
        *ELSE DEST.qword[j].byte[b] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VGF2P8AFFINEQB dest, src1, src2, imm8 (128b and 256b VEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256)
FOR j := 0 TO KL-1:
    FOR b := 0 to 7:
        DEST.qword[j].byte[b] := affine_byte(SRC2.qword[j], SRC1.qword[j].byte[b], imm8)
DEST[MAX_VL-1:VL] := 0
```

GF2P8AFFINEQB srcdest, src1, imm8 (128b SSE Encoded Version)

```
FOR j := 0 TO 1:
    FOR b := 0 to 7:
        SRCDEST.qword[j].byte[b] := affine_byte(SRC1.qword[j], SRCDEST.qword[j].byte[b], imm8)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)GF2P8AFFINEQB __m128i _mm_gf2p8affine_epi64_epi8(__m128i, __m128i, int);
(V)GF2P8AFFINEQB __m128i _mm_mask_gf2p8affine_epi64_epi8(__m128i, __mmask16, __m128i, __m128i, int);
(V)GF2P8AFFINEQB __m128i _mm_maskz_gf2p8affine_epi64_epi8(__mmask16, __m128i, __m128i, int);
VGF2P8AFFINEQB __m256i _mm256_gf2p8affine_epi64_epi8(__m256i, __m256i, int);
VGF2P8AFFINEQB __m256i _mm256_mask_gf2p8affine_epi64_epi8(__m256i, __mmask32, __m256i, __m256i, int);
VGF2P8AFFINEQB __m256i _mm256_maskz_gf2p8affine_epi64_epi8(__mmask32, __m256i, __m256i, int);
VGF2P8AFFINEQB __m512i _mm512_gf2p8affine_epi64_epi8(__m512i, __m512i, int);
VGF2P8AFFINEQB __m512i _mm512_mask_gf2p8affine_epi64_epi8(__m512i, __mmask64, __m512i, __m512i, int);
VGF2P8AFFINEQB __m512i _mm512_maskz_gf2p8affine_epi64_epi8(__mmask64, __m512i, __m512i, int);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Legacy-encoded and VEX-encoded: See Table 2-21, “Type 4 Class Exception Conditions.”

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

GF2P8MULB—Galois Field Multiply Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F38 CF /r GF2P8MULB xmm1, xmm2/m128	A	V/V	GFNI	Multiplies elements in the finite field GF(2^8).
VEX.128.66.0F38.W0 CF /r VGF2P8MULB xmm1, xmm2, xmm3/m128	B	V/V	AVX GFNI	Multiplies elements in the finite field GF(2^8).
VEX.256.66.0F38.W0 CF /r VGF2P8MULB ymm1, ymm2, ymm3/m256	B	V/V	AVX GFNI	Multiplies elements in the finite field GF(2^8).
EVEX.128.66.0F38.W0 CF /r VGF2P8MULB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL OR AVX10.1[1]) GFNI	Multiplies elements in the finite field GF(2^8).
EVEX.256.66.0F38.W0 CF /r VGF2P8MULB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL OR AVX10.1[1]) GFNI	Multiplies elements in the finite field GF(2^8).
EVEX.512.66.0F38.W0 CF /r VGF2P8MULB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	(AVX512F OR AVX10.1[1]) GFNI	Multiplies elements in the finite field GF(2^8).

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The instruction multiplies elements in the finite field GF(2^8), operating on a byte (field element) in the first source operand and the corresponding byte in a second source operand. The field GF(2^8) is represented in polynomial representation with the reduction polynomial $x^8 + x^4 + x^3 + x + 1$.

This instruction does not support broadcasting.

The EVEX encoded form of this instruction supports memory fault suppression. The SSE encoded forms of the instruction require16B alignment on their memory operations.

Operation

```
define gf2p8mul_byte(src1byte, src2byte):
    tword := 0
    FOR i := 0 to 7:
        IF src2byte.bit[i]:
            tword := tword XOR (src1byte<< i)
        * carry out polynomial reduction by the characteristic polynomial p*
    FOR i := 14 downto 8:
        p := 0x11B << (i-8)         *0x11B = 0000_0001_0001_1011 in binary*
        IF tword.bit[i]:
            tword := tword XOR p
return tword.byte[0]
```

VGF2P8MULB dest, src1, src2 (EVEX Encoded Version)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[j] := gf2p8mul_byte(SRC1.byte[j], SRC2.byte[j])
    ELSE iF *zeroing*:
        DEST.byte[j] := 0
    * ELSE DEST.byte[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VGF2P8MULB dest, src1, src2 (128b and 256b VEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256)
FOR j := 0 TO KL-1:
    DEST.byte[j] := gf2p8mul_byte(SRC1.byte[j], SRC2.byte[j])
DEST[MAX_VL-1:VL] := 0
```

GF2P8MULB srcdest, src1 (128b SSE Encoded Version)

```
FOR j := 0 TO 15:
    SRCDEST.byte[j] :=gf2p8mul_byte(SRCDEST.byte[j], SRC1.byte[j])
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)GF2P8MULB __m128i _mm_gf2p8mul_epi8(__m128i, __m128i);
(V)GF2P8MULB __m128i _mm_mask_gf2p8mul_epi8(__m128i, __mmask16, __m128i, __m128i);
(V)GF2P8MULB __m128i _mm_maskz_gf2p8mul_epi8(__mmask16, __m128i, __m128i);
VGF2P8MULB __m256i _mm256_gf2p8mul_epi8(__m256i, __m256i);
VGF2P8MULB __m256i _mm256_mask_gf2p8mul_epi8(__m256i, __mmask32, __m256i, __m256i);
VGF2P8MULB __m256i _mm256_maskz_gf2p8mul_epi8(__mmask32, __m256i, __m256i);
VGF2P8MULB __m512i _mm512_gf2p8mul_epi8(__m512i, __m512i);
VGF2P8MULB __m512i _mm512_mask_gf2p8mul_epi8(__m512i, __mmask64, __m512i, __m512i);
VGF2P8MULB __m512i _mm512_maskz_gf2p8mul_epi8(__mmask64, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Legacy-encoded and VEX-encoded: See Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded: See Table 2-49, “Type E4 Class Exception Conditions.”

INSERTPS—Insert Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 21 /r ib INSERTPS xmm1, xmm2/m32, imm8	A	V/V	SSE4_1	Insert a single precision floating-point value selected by imm8 from xmm2/m32 into xmm1 at the specified destination element specified by imm8 and zero out destination elements in xmm1 as indicated in imm8.
VEX.128.66.0F3A.WIG 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	B	V/V	AVX	Insert a single precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.
EVEX.128.66.0F3A.W0 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	C	V/V	AVX512F OR AVX10.1[1]	Insert a single precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(register source form)

Copy a single precision scalar floating-point element into a 128-bit vector register. The immediate operand has three fields, where the ZMask bits specify which elements of the destination will be set to zero, the Count_D bits specify which element of the destination will be overwritten with the scalar value, and for vector register sources the Count_S bits specify which element of the source will be copied. When the scalar source is a memory operand the Count_S bits are ignored.

(memory source form)

Load a floating-point element from a 32-bit memory location and destination operand it into the first source at the location indicated by the Count_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

128-bit Legacy SSE version: The first source register is an XMM register. The second source operand is either an XMM register or a 32-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.

VEX.128 and EVEX encoded version: The destination and first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

If VINSERTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

VINSERTPS (VEX.128 and EVEX Encoded Version)

```
IF (SRC = REG) THEN COUNT_S := imm8[7:6]
    ELSE COUNT_S := 0
COUNT_D := imm8[5:4]
ZMASK := imm8[3:0]
CASE (COUNT_S) OF
    0: TMP := SRC2[31:0]
    1: TMP := SRC2[63:32]
    2: TMP := SRC2[95:64]
    3: TMP := SRC2[127:96]
ESAC;
CASE (COUNT_D) OF
    0: TMP2[31:0] := TMP
        TMP2[127:32] := SRC1[127:32]
    1: TMP2[63:32] := TMP
        TMP2[31:0] := SRC1[31:0]
        TMP2[127:64] := SRC1[127:64]
    2: TMP2[95:64] := TMP
        TMP2[63:0] := SRC1[63:0]
        TMP2[127:96] := SRC1[127:96]
    3: TMP2[127:96] := TMP
        TMP2[95:0] := SRC1[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] := 00000000H
    ELSE DEST[31:0] := TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] := 00000000H
    ELSE DEST[63:32] := TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] := 00000000H
    ELSE DEST[95:64] := TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] := 00000000H
    ELSE DEST[127:96] := TMP2[127:96]
DEST[MAXVL-1:128] := 0
```

INSERTPS (128-bit Legacy SSE Version)

```
IF (SRC = REG) THEN COUNT_S :=imm8[7:6]
    ELSE COUNT_S :=0
COUNT_D := imm8[5:4]
ZMASK := imm8[3:0]
CASE (COUNT_S) OF
    0: TMP := SRC[31:0]
    1: TMP := SRC[63:32]
    2: TMP := SRC[95:64]
    3: TMP := SRC[127:96]
ESAC;

CASE (COUNT_D) OF
    0: TMP2[31:0] := TMP
        TMP2[127:32] := DEST[127:32]
    1: TMP2[63:32] := TMP
        TMP2[31:0] := DEST[31:0]
        TMP2[127:64] := DEST[127:64]
    2: TMP2[95:64] := TMP
```

```
            TMP2[63:0] := DEST[63:0]
            TMP2[127:96] := DEST[127:96]
    3: TMP2[127:96] := TMP
            TMP2[95:0] := DEST[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] := 00000000H
    ELSE DEST[31:0] := TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] := 00000000H
    ELSE DEST[63:32] := TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] := 00000000H
    ELSE DEST[95:64] := TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] := 00000000H
    ELSE DEST[127:96] := TMP2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VINSERTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);
INSETRTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”

KADDW/KADDB/KADDQ/KADDD—ADD Two Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 4A /r KADDW k1, k2, k3	RVR	V/V	AVX512DQ OR AVX10.1[1]	Add 16 bits masks in k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 4A /r KADDB k1, k2, k3	RVR	V/V	AVX512DQ OR AVX10.1[1]	Add 8 bits masks in k2 and k3 and place result in k1.
VEX.L1.0F.W1 4A /r KADDQ k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1[1]	Add 64 bits masks in k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 4A /r KADDD k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1[1]	Add 32 bits masks in k2 and k3 and place result in k1.

NOTES:

1. For opmask instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of supported opmask instructions available to the programmer listed in the above opcode table. Quadword opmask instructions will only be supported on processors supporting vector lengths of 512 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Adds the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

Operation

```
KADDW
DEST[15:0] := SRC1[15:0] + SRC2[15:0]
DEST[MAX_KL-1:16] := 0

KADDB
DEST[7:0] := SRC1[7:0] + SRC2[7:0]
DEST[MAX_KL-1:8] := 0

KADDQ
DEST[63:0] := SRC1[63:0] + SRC2[63:0]
DEST[MAX_KL-1:64] := 0

KADDD
DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[MAX_KL-1:32] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
KADDW __mmask16 _kadd_mask16 (__mmask16 a, __mmask16 b);
KADDB __mmask8 _kadd_mask8 (__mmask8 a, __mmask8 b);
KADDQ __mmask64 _kadd_mask64 (__mmask64 a, __mmask64 b);
KADDD __mmask32 _kadd_mask32 (__mmask32 a, __mmask32 b);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KANDW/KANDB/KANDQ/KANDD—Bitwise Logical AND Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 41 /r KANDW k1, k2, k3	RVR	V/V	AVX512F OR AVX10.1[1]	Bitwise AND 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 41 /r KANDB k1, k2, k3	RVR	V/V	AVX512DQ OR AVX10.1[1]	Bitwise AND 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 41 /r KANDQ k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1[1]	Bitwise AND 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 41 /r KANDD k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1[1]	Bitwise AND 32 bits masks k2 and k3 and place result in k1.

NOTES:

1. For opmask instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of supported opmask instructions available to the programmer listed in the above opcode table. Quadword opmask instructions will only be supported on processors supporting vector lengths of 512 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise AND between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

Operation

KANDW

```
DEST[15:0] := SRC1[15:0] BITWISE AND SRC2[15:0]
DEST[MAX_KL-1:16] := 0
```

KANDB

```
DEST[7:0] := SRC1[7:0] BITWISE AND SRC2[7:0]
DEST[MAX_KL-1:8] := 0
```

KANDQ

```
DEST[63:0] := SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[MAX_KL-1:64] := 0
```

KANDD

```
DEST[31:0] := SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[MAX_KL-1:32] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KANDW __mmask16 _mm512_kand(__mmask16 a, __mmask16 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KANDNW/KANDNB/KANDNQ/KANDND—Bitwise Logical AND NOT Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 42 /r KANDNW k1, k2, k3	RVR	V/V	AVX512F OR AVX10.1[1]	Bitwise AND NOT 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 42 /r KANDNB k1, k2, k3	RVR	V/V	AVX512DQ OR AVX10.1[1]	Bitwise AND NOT 8 bits masks k1 and k2 and place result in k1.
VEX.L1.0F.W1 42 /r KANDNQ k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1[1]	Bitwise AND NOT 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 42 /r KANDND k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1[1]	Bitwise AND NOT 32 bits masks k2 and k3 and place result in k1.

NOTES:

1. For opmask instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of supported opmask instructions available to the programmer listed in the above opcode table. Quadword opmask instructions will only be supported on processors supporting vector lengths of 512 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise AND NOT between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

Operation

KANDNW

```
DEST[15:0] := (BITWISE NOT SRC1[15:0]) BITWISE AND SRC2[15:0]
DEST[MAX_KL-1:16] := 0
```

KANDNB

```
DEST[7:0] := (BITWISE NOT SRC1[7:0]) BITWISE AND SRC2[7:0]
DEST[MAX_KL-1:8] := 0
```

KANDNQ

```
DEST[63:0] := (BITWISE NOT SRC1[63:0]) BITWISE AND SRC2[63:0]
DEST[MAX_KL-1:64] := 0
```

KANDND

```
DEST[31:0] := (BITWISE NOT SRC1[31:0]) BITWISE AND SRC2[31:0]
DEST[MAX_KL-1:32] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
KANDNW __mmask16 _mm512_kandn(__mmask16 a, __mmask16 b);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KMOVW/KMOVB/KMOVQ/KMOVD—Move From and to Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 90 /r KMOVW k1, k2/m16	RM	V/V	AVX512F OR AVX10.1[1]	Move 16 bits mask from k2/m16 and store the result in k1.
VEX.L0.66.0F.W0 90 /r KMOVB k1, k2/m8	RM	V/V	AVX512DQ OR AVX10.1[1]	Move 8 bits mask from k2/m8 and store the result in k1.
VEX.L0.0F.W1 90 /r KMOVQ k1, k2/m64	RM	V/V	AVX512BW OR AVX10.1[1]	Move 64 bits mask from k2/m64 and store the result in k1.
VEX.L0.66.0F.W1 90 /r KMOVD k1, k2/m32	RM	V/V	AVX512BW OR AVX10.1[1]	Move 32 bits mask from k2/m32 and store the result in k1.
VEX.L0.0F.W0 91 /r KMOVW m16, k1	MR	V/V	AVX512F OR AVX10.1[1]	Move 16 bits mask from k1 and store the result in m16.
VEX.L0.66.0F.W0 91 /r KMOVB m8, k1	MR	V/V	AVX512DQ OR AVX10.1[1]	Move 8 bits mask from k1 and store the result in m8.
VEX.L0.0F.W1 91 /r KMOVQ m64, k1	MR	V/V	AVX512BW OR AVX10.1[1]	Move 64 bits mask from k1 and store the result in m64.
VEX.L0.66.0F.W1 91 /r KMOVD m32, k1	MR	V/V	AVX512BW OR AVX10.1[1]	Move 32 bits mask from k1 and store the result in m32.
VEX.L0.0F.W0 92 /r KMOVW k1, r32	RR	V/V	AVX512F OR AVX10.1[1]	Move 16 bits mask from r32 to k1.
VEX.L0.66.0F.W0 92 /r KMOVB k1, r32	RR	V/V	AVX512DQ OR AVX10.1[1]	Move 8 bits mask from r32 to k1.
VEX.L0.F2.0F.W1 92 /r KMOVQ k1, r64	RR	V/I	AVX512BW OR AVX10.1[1]	Move 64 bits mask from r64 to k1.
VEX.L0.F2.0F.W0 92 /r KMOVD k1, r32	RR	V/V	AVX512BW OR AVX10.1[1]	Move 32 bits mask from r32 to k1.
VEX.L0.0F.W0 93 /r KMOVW r32, k1	RR	V/V	AVX512F OR AVX10.1[1]	Move 16 bits mask from k1 to r32.
VEX.L0.66.0F.W0 93 /r KMOVB r32, k1	RR	V/V	AVX512DQ OR AVX10.1[1]	Move 8 bits mask from k1 to r32.
VEX.L0.F2.0F.W1 93 /r KMOVQ r64, k1	RR	V/I	AVX512BW OR AVX10.1[1]	Move 64 bits mask from k1 to r64.
VEX.L0.F2.0F.W0 93 /r KMOVD r32, k1	RR	V/V	AVX512BW OR AVX10.1[1]	Move 32 bits mask from k1 to r32.

NOTES:

1. For opmask instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of supported opmask instructions available to the programmer listed in the above opcode table. Quadword opmask instructions will only be supported on processors supporting vector lengths of 512 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RM	ModRM:reg (w)	ModRM:r/m (r)
MR	ModRM:r/m (w, ModRM:[7:6] must not be 11b)	ModRM:reg (r)
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Copies values from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be mask registers, memory location or general purpose. The instruction cannot be used to transfer data between general purpose registers and or memory locations.

When moving to a mask register, the result is zero extended to MAX_KL size (i.e., 64 bits currently). When moving to a general-purpose register (GPR), the result is zero-extended to the size of the destination. In 32-bit mode, the default GPR destination's size is 32 bits. In 64-bit mode, the default GPR destination's size is 64 bits. Note that VEX.W can only be used to modify the size of the GPR operand in 64b mode.

Operation

KMOVW

```
IF *destination is a memory location*
    DEST[15:0] := SRC[15:0]
IF *destination is a mask register or a GPR *
    DEST := ZeroExtension(SRC[15:0])
```

KMOVB

```
IF *destination is a memory location*
    DEST[7:0] := SRC[7:0]
IF *destination is a mask register or a GPR *
    DEST := ZeroExtension(SRC[7:0])
```

KMOVQ

```
IF *destination is a memory location or a GPR*
    DEST[63:0] := SRC[63:0]
IF *destination is a mask register*
    DEST := ZeroExtension(SRC[63:0])
```

KMOVD

```
IF *destination is a memory location*
    DEST[31:0] := SRC[31:0]
IF *destination is a mask register or a GPR *
    DEST := ZeroExtension(SRC[31:0])
```

Intel C/C++ Compiler Intrinsic Equivalent

KMOVW __mmask16 _mm512_kmov(__mmask16 a);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Instructions with RR operand encoding, see Table 2-63, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

Instructions with RM or MR operand encoding, see Table 2-64, "TYPE K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory)."

KNOTW/KNOTB/KNOTQ/KNOTD—NOT Mask Register

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 44 /r KNOTW k1, k2	RR	V/V	AVX512F OR AVX10.1[1]	Bitwise NOT of 16 bits mask k2.
VEX.L0.66.0F.W0 44 /r KNOTB k1, k2	RR	V/V	AVX512DQ OR AVX10.1[1]	Bitwise NOT of 8 bits mask k2.
VEX.L0.0F.W1 44 /r KNOTQ k1, k2	RR	V/V	AVX512BW OR AVX10.1[1]	Bitwise NOT of 64 bits mask k2.
VEX.L0.66.0F.W1 44 /r KNOTD k1, k2	RR	V/V	AVX512BW OR AVX10.1[1]	Bitwise NOT of 32 bits mask k2.

NOTES:

1. For opmask instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of supported opmask instructions available to the programmer listed in the above opcode table. Quadword opmask instructions will only be supported on processors supporting vector lengths of 512 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise NOT of vector mask k2 and writes the result into vector mask k1.

Operation

```
KNOTW
DEST[15:0] := BITWISE NOT SRC[15:0]
DEST[MAX_KL-1:16] := 0

KNOTB
DEST[7:0] := BITWISE NOT SRC[7:0]
DEST[MAX_KL-1:8] := 0

KNOTQ
DEST[63:0] := BITWISE NOT SRC[63:0]
DEST[MAX_KL-1:64] := 0

KNOTD
DEST[31:0] := BITWISE NOT SRC[31:0]
DEST[MAX_KL-1:32] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

`KNOTW __mmask16 _mm512_knot(__mmask16 a);`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KORW/KORB/KORQ/KORD—Bitwise Logical OR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 45 /r KORW k1, k2, k3	RVR	V/V	AVX512F OR AVX10.1[1]	Bitwise OR 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 45 /r KORB k1, k2, k3	RVR	V/V	AVX512DQ OR AVX10.1[1]	Bitwise OR 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 45 /r KORQ k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1[1]	Bitwise OR 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 45 /r KORD k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1[1]	Bitwise OR 32 bits masks k2 and k3 and place result in k1.

NOTES:

1. For opmask instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of supported opmask instructions available to the programmer listed in the above opcode table. Quadword opmask instructions will only be supported on processors supporting vector lengths of 512 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise OR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation

KORW

```
DEST[15:0] := SRC1[15:0] BITWISE OR SRC2[15:0]
DEST[MAX_KL-1:16] := 0
```

KORB

```
DEST[7:0] := SRC1[7:0] BITWISE OR SRC2[7:0]
DEST[MAX_KL-1:8] := 0
```

KORQ

```
DEST[63:0] := SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[MAX_KL-1:64] := 0
```

KORD

```
DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[MAX_KL-1:32] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KORW __mmask16 _mm512_kor(__mmask16 a, __mmask16 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KORTESTW/KORTESTB/KORTESTQ/KORTESTD—OR Masks and Set Flags

Opcode/ Instruction	Op/E n	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 98 /r KORTESTW k1, k2	RR	V/V	AVX512F OR AVX10.1[1]	Bitwise OR 16 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.66.0F.W0 98 /r KORTESTB k1, k2	RR	V/V	AVX512DQ OR AVX10.1[1]	Bitwise OR 8 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.0F.W1 98 /r KORTESTQ k1, k2	RR	V/V	AVX512BW OR AVX10.1[1]	Bitwise OR 64 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.66.0F.W1 98 /r KORTESTD k1, k2	RR	V/V	AVX512BW OR AVX10.1[1]	Bitwise OR 32 bits masks k1 and k2 and update ZF and CF accordingly.

NOTES:

1. For opmask instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of supported opmask instructions available to the programmer listed in the above opcode table. Quadword opmask instructions will only be supported on processors supporting vector lengths of 512 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise OR between the vector mask register k2, and the vector mask register k1, and sets CF and ZF based on the operation result.

ZF flag is set if both sources are 0x0. CF is set if, after the OR operation is done, the operation result is all 1's.

Operation

KORTESTW

```
TMP[15:0] := DEST[15:0] BITWISE OR SRC[15:0]
IF(TMP[15:0]=0)
    THEN ZF := 1
    ELSE ZF := 0
FI;
IF(TMP[15:0]=FFFFh)
    THEN CF := 1
    ELSE CF := 0
FI;
```

KORTESTB

```
TMP[7:0] := DEST[7:0] BITWISE OR SRC[7:0]
IF(TMP[7:0]=0)
    THEN ZF := 1
    ELSE ZF := 0
FI;
IF(TMP[7:0]==FFh)
    THEN CF := 1
    ELSE CF := 0

FI;
```

KORTESTQ

```
TMP[63:0] := DEST[63:0] BITWISE OR SRC[63:0]
IF(TMP[63:0]=0)
    THEN ZF := 1
    ELSE ZF := 0
FI;
IF(TMP[63:0]==FFFFFFFF_FFFFFFFFh)
    THEN CF := 1
    ELSE CF := 0
FI;
```

KORTESTD

```
TMP[31:0] := DEST[31:0] BITWISE OR SRC[31:0]
IF(TMP[31:0]=0)
    THEN ZF := 1
    ELSE ZF := 0
FI;
IF(TMP[31:0]=FFFFFFFFh)
    THEN CF := 1
    ELSE CF := 0
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

KORTESTW __mmask16 _mm512_kortest[cz](__mmask16 a, __mmask16 b);

Flags Affected

The ZF flag is set if the result of OR-ing both sources is all 0s.

The CF flag is set if the result of OR-ing both sources is all 1s.

The OF, SF, AF, and PF flags are set to 0.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KSHIFTLW/KSHIFTLB/KSHIFTLQ/KSHIFTLD—Shift Left Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.66.0F3A.W1 32 /r KSHIFTLW k1, k2, imm8	RRI	V/V	AVX512F OR AVX10.1[1]	Shift left 16 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 32 /r KSHIFTLB k1, k2, imm8	RRI	V/V	AVX512DQ OR AVX10.1[1]	Shift left 8 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W1 33 /r KSHIFTLQ k1, k2, imm8	RRI	V/V	AVX512BW OR AVX10.1[1]	Shift left 64 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 33 /r KSHIFTLD k1, k2, imm8	RRI	V/V	AVX512BW OR AVX10.1[1]	Shift left 32 bits in k2 by immediate and write result in k1.

NOTES:

1. For opmask instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of supported opmask instructions available to the programmer listed in the above opcode table. Quadword opmask instructions will only be supported on processors supporting vector lengths of 512 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RRI	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)	imm8

Description

Shifts 8/16/32/64 bits in the second operand (source operand) left by the count specified in immediate byte and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

Operation

KSHIFTLW

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=15
    THEN DEST[15:0] := SRC1[15:0] << COUNT;
FI;
```

KSHIFTLB

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=7
        THEN    DEST[7:0] := SRC1[7:0] << COUNT;
FI;
```

KSHIFTLQ

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=63
        THEN    DEST[63:0] := SRC1[63:0] << COUNT;
FI;
```

KSHIFTLD

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=31
        THEN    DEST[31:0] := SRC1[31:0] << COUNT;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

Compiler auto generates KSHIFTLW when needed.

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KSHIFTRW/KSHIFTRB/KSHIFTRQ/KSHIFTRD—Shift Right Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.66.0F3A.W1 30 /r KSHIFTRW k1, k2, imm8	RRI	V/V	AVX512F OR AVX10.1[1]	Shift right 16 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 30 /r KSHIFTRB k1, k2, imm8	RRI	V/V	AVX512DQ OR AVX10.1[1]	Shift right 8 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W1 31 /r KSHIFTRQ k1, k2, imm8	RRI	V/V	AVX512BW OR AVX10.1[1]	Shift right 64 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 31 /r KSHIFTRD k1, k2, imm8	RRI	V/V	AVX512BW OR AVX10.1[1]	Shift right 32 bits in k2 by immediate and write result in k1.

NOTES:

1. For opmask instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of supported opmask instructions available to the programmer listed in the above opcode table. Quadword opmask instructions will only be supported on processors supporting vector lengths of 512 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RRI	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)	imm8

Description

Shifts 8/16/32/64 bits in the second operand (source operand) right by the count specified in immediate and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

Operation

KSHIFTRW

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=15
    THEN DEST[15:0] := SRC1[15:0] >> COUNT;
FI;
```

KSHIFTRB

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=7
        THEN    DEST[7:0] := SRC1[7:0] >> COUNT;
FI;
```

KSHIFTRQ

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=63
        THEN    DEST[63:0] := SRC1[63:0] >> COUNT;
FI;
```

KSHIFTRD

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=31
        THEN    DEST[31:0] := SRC1[31:0] >> COUNT;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

Compiler auto generates KSHIFTRW when needed.

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KTESTW/KTESTB/KTESTQ/KTESTD—Packed Bit Test Masks and Set Flags

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 99 /r KTESTW k1, k2	RR	V/V	AVX512DQ OR AVX10.1[1]	Set ZF and CF depending on sign bit AND and ANDN of 16 bits mask register sources.
VEX.L0.66.0F.W0 99 /r KTESTB k1, k2	RR	V/V	AVX512DQ OR AVX10.1[1]	Set ZF and CF depending on sign bit AND and ANDN of 8 bits mask register sources.
VEX.L0.0F.W1 99 /r KTESTQ k1, k2	RR	V/V	AVX512BW OR AVX10.1[1]	Set ZF and CF depending on sign bit AND and ANDN of 64 bits mask register sources.
VEX.L0.66.0F.W1 99 /r KTESTD k1, k2	RR	V/V	AVX512BW OR AVX10.1[1]	Set ZF and CF depending on sign bit AND and ANDN of 32 bits mask register sources.

NOTES:

1. For opmask instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of supported opmask instructions available to the programmer listed in the above opcode table. Quadword opmask instructions will only be supported on processors supporting vector lengths of 512 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RR	ModRM:reg (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise comparison of the bits of the first source operand and corresponding bits in the second source operand. If the AND operation produces all zeros, the ZF is set else the ZF is clear. If the bitwise AND operation of the inverted first source operand with the second source operand produces all zeros the CF is set else the CF is clear. Only the EFLAGS register is updated.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

KTESTW

```
TEMP[15:0] := SRC2[15:0] AND SRC1[15:0]
IF (TEMP[15:0] = = 0)
    THEN ZF :=1;
    ELSE ZF := 0;
FI;
TEMP[15:0] := SRC2[15:0] AND NOT SRC1[15:0]
IF (TEMP[15:0] = = 0)
    THEN CF :=1;
    ELSE CF := 0;
FI;
AF := OF := PF := SF := 0;
```

```
KTESTB
TEMP[7:0] := SRC2[7:0] AND SRC1[7:0]
IF (TEMP[7:0] = = 0)
    THEN ZF :=1;
    ELSE ZF := 0;
FI;
TEMP[7:0] := SRC2[7:0] AND NOT SRC1[7:0]
IF (TEMP[7:0] = = 0)
    THEN CF :=1;
    ELSE CF := 0;
FI;
AF := OF := PF := SF := 0;

KTESTQ
TEMP[63:0] := SRC2[63:0] AND SRC1[63:0]
IF (TEMP[63:0] = = 0)
    THEN ZF :=1;
    ELSE ZF := 0;
FI;
TEMP[63:0] := SRC2[63:0] AND NOT SRC1[63:0]
IF (TEMP[63:0] = = 0)
    THEN CF :=1;
    ELSE CF := 0;
FI;
AF := OF := PF := SF := 0;

KTESTD
TEMP[31:0] := SRC2[31:0] AND SRC1[31:0]
IF (TEMP[31:0] = = 0)
    THEN ZF :=1;
    ELSE ZF := 0;
FI;
TEMP[31:0] := SRC2[31:0] AND NOT SRC1[31:0]
IF (TEMP[31:0] = = 0)
    THEN CF :=1;
    ELSE CF := 0;
FI;
AF := OF := PF := SF := 0;
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KUNPCKBW/KUNPCKWD/KUNPCKDQ—Unpack for Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.66.0F.W0 4B /r KUNPCKBW k1, k2, k3	RVR	V/V	AVX512F OR AVX10.1[1]	Unpack 8-bit masks in k2 and k3 and write word result in k1.
VEX.L1.0F.W0 4B /r KUNPCKWD k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1[1]	Unpack 16-bit masks in k2 and k3 and write doubleword result in k1.
VEX.L1.0F.W1 4B /r KUNPCKDQ k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1[1]	Unpack 32-bit masks in k2 and k3 and write quadword result in k1.

NOTES:

1. For opmask instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of supported opmask instructions available to the programmer listed in the above opcode table. Quadword opmask instructions will only be supported on processors supporting vector lengths of 512 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Unpacks the lower 8/16/32 bits of the second and third operands (source operands) into the low part of the first operand (destination operand), starting from the low bytes. The result is zero-extended in the destination.

Operation

KUNPCKBW

```
DEST[7:0] := SRC2[7:0]
DEST[15:8] := SRC1[7:0]
DEST[MAX_KL-1:16] := 0
```

KUNPCKWD

```
DEST[15:0] := SRC2[15:0]
DEST[31:16] := SRC1[15:0]
DEST[MAX_KL-1:32] := 0
```

KUNPCKDQ

```
DEST[31:0] := SRC2[31:0]
DEST[63:32] := SRC1[31:0]
DEST[MAX_KL-1:64] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KUNPCKBW __mmask16 _mm512_kunpackb(__mmask16 a, __mmask16 b);
KUNPCKDQ __mmask64 _mm512_kunpackd(__mmask64 a, __mmask64 b);
KUNPCKWD __mmask32 _mm512_kunpackw(__mmask32 a, __mmask32 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KXNORW/KXNORB/KXNORQ/KXNORD—Bitwise Logical XNOR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 46 /r KXNORW k1, k2, k3	RVR	V/V	AVX512F OR AVX10.1[1]	Bitwise XNOR 16-bit masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 46 /r KXNORB k1, k2, k3	RVR	V/V	AVX512DQ OR AVX10.1[1]	Bitwise XNOR 8-bit masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 46 /r KXNORQ k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1[1]	Bitwise XNOR 64-bit masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 46 /r KXNORD k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1[1]	Bitwise XNOR 32-bit masks k2 and k3 and place result in k1.

NOTES:

1. For opmask instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of supported opmask instructions available to the programmer listed in the above opcode table. Quadword opmask instructions will only be supported on processors supporting vector lengths of 512 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise XNOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation

KXNORW

```
DEST[15:0] := NOT (SRC1[15:0] BITWISE XOR SRC2[15:0])
DEST[MAX_KL-1:16] := 0
```

KXNORB

```
DEST[7:0] := NOT (SRC1[7:0] BITWISE XOR SRC2[7:0])
DEST[MAX_KL-1:8] := 0
```

KXNORQ

```
DEST[63:0] := NOT (SRC1[63:0] BITWISE XOR SRC2[63:0])
DEST[MAX_KL-1:64] := 0
```

KXNORD

```
DEST[31:0] := NOT (SRC1[31:0] BITWISE XOR SRC2[31:0])
DEST[MAX_KL-1:32] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KXNORW __mmask16 _mm512_kxnor(__mmask16 a, __mmask16 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KXORW/KXORB/KXORQ/KXORD—Bitwise Logical XOR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 47 /r KXORW k1, k2, k3	RVR	V/V	AVX512F OR AVX10.1[1]	Bitwise XOR 16-bit masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 47 /r KXORB k1, k2, k3	RVR	V/V	AVX512DQ OR AVX10.1[1]	Bitwise XOR 8-bit masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 47 /r KXORQ k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1[1]	Bitwise XOR 64-bit masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 47 /r KXORD k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1[1]	Bitwise XOR 32-bit masks k2 and k3 and place result in k1.

NOTES:

1. For opmask instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of supported opmask instructions available to the programmer listed in the above opcode table. Quadword opmask instructions will only be supported on processors supporting vector lengths of 512 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise XOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation

KXORW

```
DEST[15:0] := SRC1[15:0] BITWISE XOR SRC2[15:0]
DEST[MAX_KL-1:16] := 0
```

KXORB

```
DEST[7:0] := SRC1[7:0] BITWISE XOR SRC2[7:0]
DEST[MAX_KL-1:8] := 0
```

KXORQ

```
DEST[63:0] := SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[MAX_KL-1:64] := 0
```

KXORD

```
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[MAX_KL-1:32] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

`KXORW __mmask16 _mm512_kxor(__mmask16 a, __mmask16 b);`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

MAXPD—Maximum of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5F /r MAXPD xmm1, xmm2/m128	A	V/V	SSE2	Return the maximum double precision floating-point values between xmm1 and xmm2/m128.
VEX.128.66.0F.WIG 5F /r VMAXPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the maximum double precision floating-point values between xmm2 and xmm3/m128.
VEX.256.66.0F.WIG 5F /r VMAXPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum packed double precision floating-point values between ymm2 and ymm3/m256.
EVEX.128.66.0F.W1 5F /r VMAXPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Return the maximum packed double precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 5F /r VMAXPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Return the maximum packed double precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 5F /r VMAXPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	C	V/V	AVX512F OR AVX10.1[1]	Return the maximum packed double precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed double precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPD can be emulated using a sequence of instructions, such as a comparison followed by AND, ANDN, and OR.

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

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. 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 ZMM register destination are unmodified.

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
         ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
         ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
         ELSE IF (SRC1 > SRC2) THEN DEST := SRC1;
         ELSE DEST := SRC2;
    FI;
}
```

VMAXPD (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] := MAX(SRC1[i+63:i], SRC2[63:0])
                   ELSE
                        DEST[i+63:i] := MAX(SRC1[i+63:i], SRC2[i+63:i])
              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
```

VMAXPD (VEX.256 Encoded Version)

```
DEST[63:0] := MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] := MAX(SRC1[127:64], SRC2[127:64])
DEST[191:128] := MAX(SRC1[191:128], SRC2[191:128])
DEST[255:192] := MAX(SRC1[255:192], SRC2[255:192])
DEST[MAXVL-1:256] := 0
```

VMAXPD (VEX.128 Encoded Version)

```
DEST[63:0] := MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] := MAX(SRC1[127:64], SRC2[127:64])
DEST[MAXVL-1:128] := 0
```

MAXPD (128-bit Legacy SSE Version)

```
DEST[63:0] := MAX(DEST[63:0], SRC[63:0])
DEST[127:64] := MAX(DEST[127:64], SRC[127:64])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMAXPD __m512d _mm512_max_pd( __m512d a, __m512d b);
VMAXPD __m512d _mm512_mask_max_pd(__m512d s, __mmask8 k, __m512d a, __m512d b,);
VMAXPD __m512d _mm512_maskz_max_pd( __mmask8 k, __m512d a, __m512d b);
VMAXPD __m512d _mm512_max_round_pd( __m512d a, __m512d b, int);
VMAXPD __m512d _mm512_mask_max_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMAXPD __m512d _mm512_maskz_max_round_pd( __mmask8 k, __m512d a, __m512d b, int);
VMAXPD __m256d _mm256_mask_max_pd(__m5256d s, __mmask8 k, __m256d a, __m256d b);
VMAXPD __m256d _mm256_maskz_max_pd( __mmask8 k, __m256d a, __m256d b);
VMAXPD __m128d _mm_mask_max_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMAXPD __m128d _mm_maskz_max_pd( __mmask8 k, __m128d a, __m128d b);
VMAXPD __m256d _mm256_max_pd (__m256d a, __m256d b);
(V)MAXPD __m128d _mm_max_pd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

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

MAXPS—Maximum of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5F /r MAXPS xmm1, xmm2/m128	A	V/V	SSE	Return the maximum single precision floating-point values between xmm1 and xmm2/mem.
VEX.128.0F.WIG 5F /r VMAXPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the maximum single precision floating-point values between xmm2 and xmm3/mem.
VEX.256.0F.WIG 5F /r VMAXPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum single precision floating-point values between ymm2 and ymm3/mem.
EVEX.128.0F.W0 5F /r VMAXPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Return the maximum packed single precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.256.0F.W0 5F /r VMAXPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Return the maximum packed single precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.512.0F.W0 5F /r VMAXPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	C	V/V	AVX512F OR AVX10.1[1]	Return the maximum packed single precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed single precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

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

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. 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 ZMM register destination are unmodified.

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
        ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST := SRC1;
        ELSE DEST := SRC2;
    FI;
}
```

VMAXPS (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] := MAX(SRC1[i+31:i], SRC2[31:0])
                ELSE
                    DEST[i+31:i] := MAX(SRC1[i+31:i], SRC2[i+31:i])
            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
```

VMAXPS (VEX.256 Encoded Version)

```
DEST[31:0] := MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] := MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] := MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] := MAX(SRC1[127:96], SRC2[127:96])
DEST[159:128] := MAX(SRC1[159:128], SRC2[159:128])
DEST[191:160] := MAX(SRC1[191:160], SRC2[191:160])
DEST[223:192] := MAX(SRC1[223:192], SRC2[223:192])
DEST[255:224] := MAX(SRC1[255:224], SRC2[255:224])
DEST[MAXVL-1:256] := 0
```

VMAXPS (VEX.128 Encoded Version)

```
DEST[31:0] := MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] := MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] := MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] := MAX(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] := 0
```

MAXPS (128-bit Legacy SSE Version)

```
DEST[31:0] := MAX(DEST[31:0], SRC[31:0])
DEST[63:32] := MAX(DEST[63:32], SRC[63:32])
DEST[95:64] := MAX(DEST[95:64], SRC[95:64])
DEST[127:96] := MAX(DEST[127:96], SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMAXPS __m512 _mm512_max_ps( __m512 a, __m512 b);
VMAXPS __m512 _mm512_mask_max_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMAXPS __m512 _mm512_maskz_max_ps( __mmask16 k, __m512 a, __m512 b);
VMAXPS __m512 _mm512_max_round_ps( __m512 a, __m512 b, int);
VMAXPS __m512 _mm512_mask_max_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMAXPS __m512 _mm512_maskz_max_round_ps( __mmask16 k, __m512 a, __m512 b, int);
VMAXPS __m256 _mm256_mask_max_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMAXPS __m256 _mm256_maskz_max_ps( __mmask8 k, __m256 a, __m256 b);
VMAXPS __m128 _mm_mask_max_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMAXPS __m128 _mm_maskz_max_ps( __mmask8 k, __m128 a, __m128 b);
VMAXPS __m256 _mm256_max_ps (__m256 a, __m256 b);
MAXPS __m128 _mm_max_ps (__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

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

MAXSD—Return Maximum Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5F /r MAXSD xmm1, xmm2/m64	A	V/V	SSE2	Return the maximum scalar double precision floating-point value between xmm2/m64 and xmm1.
VEX.LIG.F2.0F.WIG 5F /r VMAXSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the maximum scalar double precision floating-point value between xmm3/m64 and xmm2.
EVEX.LLIG.F2.0F.W1 5F /r VMAXSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	C	V/V	AVX512F OR AVX10.1[1]	Return the maximum scalar double precision floating-point value between xmm3/m64 and xmm2.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Compares the low double precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low quadword of the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. When the second source operand is a memory operand, only 64 bits are accessed.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN of either source operand be returned, the action of MAXSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: 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 encoded version: The low quadword element of the destination operand is updated according to the write-mask.

Software should ensure VMAXSD is encoded with VEX.L=0. Encoding VMAXSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
         ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
         ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
         ELSE IF (SRC1 > SRC2) THEN DEST := SRC1;
         ELSE DEST := SRC2;
    FI;
}
```

VMAXSD (EVEX Encoded Version)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := MAX(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
```

VMAXSD (VEX.128 Encoded Version)

```
DEST[63:0] := MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

MAXSD (128-bit Legacy SSE Version)

```
DEST[63:0] := MAX(DEST[63:0], SRC[63:0])
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMAXSD __m128d _mm_max_round_sd( __m128d a, __m128d b, int);
VMAXSD __m128d _mm_mask_max_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMAXSD __m128d _mm_maskz_max_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MAXSD __m128d _mm_max_sd(__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

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

MAXSS—Return Maximum Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5F /r MAXSS xmm1, xmm2/m32	A	V/V	SSE	Return the maximum scalar single precision floating-point value between xmm2/m32 and xmm1.
VEX.LIG.F3.0F.WIG 5F /r VMAXSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Return the maximum scalar single precision floating-point value between xmm3/m32 and xmm2.
EVEX.LLIG.F3.0F.W0 5F /r VMAXSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F OR AVX10.1[1]	Return the maximum scalar single precision floating-point value between xmm3/m32 and xmm2.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Compares the low single precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN from either source operand be returned, the action of MAXSS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL:128) of the destination register are zeroed.

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

Software should ensure VMAXSS is encoded with VEX.L=0. Encoding VMAXSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
        ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST := SRC1;
        ELSE DEST := SRC2;
    FI;
}
```

VMAXSS (EVEX Encoded Version)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := MAX(SRC1[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] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VMAXSS (VEX.128 Encoded Version)

```
DEST[31:0] := MAX(SRC1[31:0], SRC2[31:0])
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

MAXSS (128-bit Legacy SSE Version)

```
DEST[31:0] := MAX(DEST[31:0], SRC[31:0])
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMAXSS __m128 _mm_max_round_ss(__m128 a, __m128 b, int);
VMAXSS __m128 _mm_mask_max_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMAXSS __m128 _mm_maskz_max_round_ss(__mmask8 k, __m128 a, __m128 b, int);
MAXSS __m128 _mm_max_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

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

MINPD—Minimum of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5D /r MINPD xmm1, xmm2/m128	A	V/V	SSE2	Return the minimum double precision floating-point values between xmm1 and xmm2/mem
VEX.128.66.0F.WIG 5D /r VMINPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the minimum double precision floating-point values between xmm2 and xmm3/mem.
VEX.256.66.0F.WIG 5D /r VMINPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum packed double precision floating-point values between ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 5D /r VMINPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Return the minimum packed double precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 5D /r VMINPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Return the minimum packed double precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 5D /r VMINPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	C	V/V	AVX512F OR AVX10.1[1]	Return the minimum packed double precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed double precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

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

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. 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 ZMM register destination are unmodified.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
        ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST := SRC1;
        ELSE DEST := SRC2;
    FI;
}
```

VMINPD (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] := MIN(SRC1[i+63:i], SRC2[63:0])
                ELSE
                    DEST[i+63:i] := MIN(SRC1[i+63:i], SRC2[i+63:i])
            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
```

VMINPD (VEX.256 Encoded Version)

```
DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] := MIN(SRC1[127:64], SRC2[127:64])
DEST[191:128] := MIN(SRC1[191:128], SRC2[191:128])
DEST[255:192] := MIN(SRC1[255:192], SRC2[255:192])
```

VMINPD (VEX.128 Encoded Version)

```
DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] := MIN(SRC1[127:64], SRC2[127:64])
DEST[MAXVL-1:128] := 0
```

MINPD (128-bit Legacy SSE Version)

```
DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] := MIN(SRC1[127:64], SRC2[127:64])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINPD __m512d _mm512_min_pd( __m512d a, __m512d b);
VMINPD __m512d _mm512_mask_min_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMINPD __m512d _mm512_maskz_min_pd( __mmask8 k, __m512d a, __m512d b);
VMINPD __m512d _mm512_min_round_pd( __m512d a, __m512d b, int);
VMINPD __m512d _mm512_mask_min_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMINPD __m512d _mm512_maskz_min_round_pd( __mmask8 k, __m512d a, __m512d b, int);
VMINPD __m256d _mm256_mask_min_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VMINPD __m256d _mm256_maskz_min_pd( __mmask8 k, __m256d a, __m256d b);
VMINPD __m128d _mm_mask_min_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMINPD __m128d _mm_maskz_min_pd( __mmask8 k, __m128d a, __m128d b);
VMINPD __m256d _mm256_min_pd (__m256d a, __m256d b);
MINPD __m128d _mm_min_pd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

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

MINPS—Minimum of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5D /r MINPS xmm1, xmm2/m128	A	V/V	SSE	Return the minimum single precision floating-point values between xmm1 and xmm2/mem.
VEX.128.0F.WIG 5D /r VMINPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the minimum single precision floating-point values between xmm2 and xmm3/mem.
VEX.256.0F.WIG 5D /r VMINPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum single double precision floating-point values between ymm2 and ymm3/mem.
EVEX.128.0F.W0 5D /r VMINPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Return the minimum packed single precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.256.0F.W0 5D /r VMINPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Return the minimum packed single precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.512.0F.W0 5D /r VMINPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	C	V/V	AVX512F OR AVX10.1[1]	Return the minimum packed single precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed single precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

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

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. 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 ZMM register destination are unmodified.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
        ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST := SRC1;
        ELSE DEST := SRC2;
    FI;
}
```

VMINPS (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] := MIN(SRC1[i+31:i], SRC2[31:0])
                ELSE
                    DEST[i+31:i] := MIN(SRC1[i+31:i], SRC2[i+31:i])
            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
```

VMINPS (VEX.256 Encoded Version)

```
DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] := MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] := MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] := MIN(SRC1[127:96], SRC2[127:96])
DEST[159:128] := MIN(SRC1[159:128], SRC2[159:128])
DEST[191:160] := MIN(SRC1[191:160], SRC2[191:160])
DEST[223:192] := MIN(SRC1[223:192], SRC2[223:192])
DEST[255:224] := MIN(SRC1[255:224], SRC2[255:224])
```

VMINPS (VEX.128 Encoded Version)

```
DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] := MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] := MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] := MIN(SRC1[127:96], SRC2[127:96])
```

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

MINPS (128-bit Legacy SSE Version)

```
DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] := MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] := MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] := MIN(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINPS __m512 _mm512_min_ps( __m512 a, __m512 b);
VMINPS __m512 _mm512_mask_min_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMINPS __m512 _mm512_maskz_min_ps( __mmask16 k, __m512 a, __m512 b);
VMINPS __m512 _mm512_min_round_ps( __m512 a, __m512 b, int);
VMINPS __m512 _mm512_mask_min_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMINPS __m512 _mm512_maskz_min_round_ps( __mmask16 k, __m512 a, __m512 b, int);
VMINPS __m256 _mm256_mask_min_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMINPS __m256 _mm256_maskz_min_ps( __mmask8 k, __m256 a, __m25 b);
VMINPS __m128 _mm_mask_min_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMINPS __m128 _mm_maskz_min_ps( __mmask8 k, __m128 a, __m128 b);
VMINPS __m256 _mm256_min_ps (__m256 a, __m256 b);
MINPS __m128 _mm_min_ps (__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

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

MINSD—Return Minimum Scalar Double Precision Floating-Point Value

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
F2 0F 5D /r MINSD xmm1, xmm2/m64	A	V/V	SSE2	Return the minimum scalar double precision floating-point value between xmm2/m64 and xmm1.
VEX.LIG.F2.0F.WIG 5D /r VMINSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the minimum scalar double precision floating-point value between xmm3/m64 and xmm2.
EVEX.LLIG.F2.0F.W1 5D /r VMINSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	C	V/V	AVX512F OR AVX10.1[1]	Return the minimum scalar double precision floating-point value between xmm3/m64 and xmm2.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Compares the low double precision floating-point values in the first source operand and the second source operand, and returns the minimum value to the low quadword of the destination operand. When the source operand is a memory operand, only the 64 bits are accessed.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, then SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second source) be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: 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 encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMINSD is encoded with VEX.L=0. Encoding VMINSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
        ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST := SRC1;
        ELSE DEST := SRC2;
    FI;
}
```

MINSD (EVEX Encoded Version)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := MIN(SRC1[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] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

MINSD (VEX.128 Encoded Version)

```
DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

MINSD (128-bit Legacy SSE Version)

```
DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINSD __m128d _mm_min_round_sd(__m128d a, __m128d b, int);
VMINSD __m128d _mm_mask_min_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMINSD __m128d _mm_maskz_min_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MINSD __m128d _mm_min_sd(__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

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

MINSS—Return Minimum Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5D /r MINSS xmm1,xmm2/m32	A	V/V	SSE	Return the minimum scalar single precision floating-point value between xmm2/m32 and xmm1.
VEX.LIG.F3.0F.WIG 5D /r VMINSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Return the minimum scalar single precision floating-point value between xmm3/m32 and xmm2.
EVEX.LLIG.F3.0F.W0 5D /r VMINSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F OR AVX10.1[1]	Return the minimum scalar single precision floating-point value between xmm3/m32 and xmm2.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Compares the low single precision floating-point values in the first source operand and the second source operand and returns the minimum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN in either source operand be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by (E)VEX.vvvv. 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 encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMINSS is encoded with VEX.L=0. Encoding VMINSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
        ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST := SRC1;
        ELSE DEST := SRC2;
    FI;
}
```

MINSS (EVEX Encoded Version)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := MIN(SRC1[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] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VMINSS (VEX.128 Encoded Version)

```
DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

MINSS (128-bit Legacy SSE Version)

```
DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINSS __m128 _mm_min_round_ss( __m128 a, __m128 b, int);
VMINSS __m128 _mm_mask_min_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMINSS __m128 _mm_maskz_min_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MINSS __m128 _mm_min_ss(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

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

MOVAPD—Move Aligned Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 28 /r MOVAPD xmm1, xmm2/m128	A	V/V	SSE2	Move aligned packed double precision floating-point values from xmm2/mem to xmm1.
66 0F 29 /r MOVAPD xmm2/m128, xmm1	B	V/V	SSE2	Move aligned packed double precision floating-point values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 28 /r VMOVAPD xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed double precision floating-point values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 29 /r VMOVAPD xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed double precision floating-point values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 28 /r VMOVAPD ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed double precision floating-point values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 29 /r VMOVAPD ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed double precision floating-point values from ymm1 to ymm2/mem.
EVEX.128.66.0F.W1 28 /r VMOVAPD xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed double precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 28 /r VMOVAPD ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed double precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 28 /r VMOVAPD zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1[1]	Move aligned packed double precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 29 /r VMOVAPD xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed double precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 29 /r VMOVAPD ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed double precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 29 /r VMOVAPD zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1[1]	Move aligned packed double precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Moves 2, 4 or 8 double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-

bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit versions), 32-byte (256-bit version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX encoded versions, the operand must be aligned to the size of the memory operand. To move double precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

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

EVEX.512 encoded version:

Moves 512 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float64 memory location, to store the contents of a ZMM register into a 512-bit float64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

VEX.256 and EVEX.256 encoded versions:

Moves 256 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move double precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit versions:

Moves 128 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

(E)VEX.128 encoded version: Bits (MAXVL-1:128) of the destination ZMM register destination are zeroed.

Operation

VMOVAPD (EVEX Encoded Versions, Register-Copy Form)

```
(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[i+63:i]
        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
```

VMOVAPD (EVEX Encoded Versions, Store-Form)

```
(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[i+63:i]
        ELSE
        ELSE *DEST[i+63:i] remains unchanged*          ; merging-masking

    FI;
ENDFOR;
```

VMOVAPD (EVEX Encoded Versions, Load-Form)

```
(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[i+63:i]
        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
```

VMOVAPD (VEX.256 Encoded Version, Load - and Register Copy)

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

VMOVAPD (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVAPD (VEX.128 Encoded Version, Load - and Register Copy)

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

MOVAPD (128-bit Load- and Register-Copy- Form Legacy SSE Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVAPD (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVAPD __m512d _mm512_load_pd(void * m);
VMOVAPD __m512d _mm512_mask_load_pd(__m512d s, __mmask8 k, void * m);
VMOVAPD __m512d _mm512_maskz_load_pd(__mmask8 k, void * m);
VMOVAPD void _mm512_store_pd(void * d, __m512d a);
VMOVAPD void _mm512_mask_store_pd(void * d, __mmask8 k, __m512d a);
VMOVAPD __m256d _mm256_mask_load_pd(__m256d s, __mmask8 k, void * m);
VMOVAPD __m256d _mm256_maskz_load_pd(__mmask8 k, void * m);
VMOVAPD void _mm256_mask_store_pd(void * d, __mmask8 k, __m256d a);
VMOVAPD __m128d _mm_mask_load_pd(__m128d s, __mmask8 k, void * m);
VMOVAPD __m128d _mm_maskz_load_pd(__mmask8 k, void * m);
VMOVAPD void _mm_mask_store_pd(void * d, __mmask8 k, __m128d a);
MOVAPD __m256d _mm256_load_pd (double * p);
MOVAPD void _mm256_store_pd(double * p, __m256d a);
MOVAPD __m128d _mm_load_pd (double * p);
MOVAPD void _mm_store_pd(double * p, __m128d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, “Type 1 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-44, “Type E1 Class Exception Conditions.”
Additionally:

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

MOVAPS—Move Aligned Packed Single Precision Floating-Point Values

Opcode/ Instruction	**Op/En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 28 /r MOVAPS xmm1, xmm2/m128	A	V/V	SSE	Move aligned packed single precision floating-point values from xmm2/mem to xmm1.
NP 0F 29 /r MOVAPS xmm2/m128, xmm1	B	V/V	SSE	Move aligned packed single precision floating-point values from xmm1 to xmm2/mem.
VEX.128.0F.WIG 28 /r VMOVAPS xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed single precision floating-point values from xmm2/mem to xmm1.
VEX.128.0F.WIG 29 /r VMOVAPS xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed single precision floating-point values from xmm1 to xmm2/mem.
VEX.256.0F.WIG 28 /r VMOVAPS ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed single precision floating-point values from ymm2/mem to ymm1.
VEX.256.0F.WIG 29 /r VMOVAPS ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed single precision floating-point values from ymm1 to ymm2/mem.
EVEX.128.0F.W0 28 /r VMOVAPS xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed single precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.0F.W0 28 /r VMOVAPS ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed single precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.0F.W0 28 /r VMOVAPS zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1[1]	Move aligned packed single precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.0F.W0 29 /r VMOVAPS xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed single precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.0F.W0 29 /r VMOVAPS ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed single precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.0F.W0 29 /r VMOVAPS zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1[1]	Move aligned packed single precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Moves 4, 8 or 16 single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-

bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX.512 encoded versions, the operand must be aligned to the size of the memory operand. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

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

EVEX.512 encoded version:

Moves 512 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into a float32 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

VEX.256 and EVEX.256 encoded version:

Moves 256 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated.

128-bit versions:

Moves 128 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

(E)VEX.128 encoded version: Bits (MAXVL-1:128) of the destination ZMM register are zeroed.

Operation

VMOVAPS (EVEX Encoded Versions, Register-Copy Form)

```
(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[i+31:i]
        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
```

VMOVAPS (EVEX Encoded Versions, Store Form)

```
(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[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR;
```

VMOVAPS (EVEX Encoded Versions, Load Form)

```
(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[i+31:i]
        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
```

VMOVAPS (VEX.256 Encoded Version, Load - and Register Copy)

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

VMOVAPS (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVAPS (VEX.128 Encoded Version, Load - and Register Copy)

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

MOVAPS (128-bit Load- and Register-Copy- Form Legacy SSE Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVAPS (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVAPS __m512 _mm512_load_ps( void * m);
VMOVAPS __m512 _mm512_mask_load_ps(__m512 s, __mmask16 k, void * m);
VMOVAPS __m512 _mm512_maskz_load_ps( __mmask16 k, void * m);
VMOVAPS void _mm512_store_ps( void * d, __m512 a);
VMOVAPS void _mm512_mask_store_ps( void * d, __mmask16 k, __m512 a);
VMOVAPS __m256 _mm256_mask_load_ps(__m256 a, __mmask8 k, void * s);
VMOVAPS __m256 _mm256_maskz_load_ps( __mmask8 k, void * s);
VMOVAPS void _mm256_mask_store_ps( void * d, __mmask8 k, __m256 a);
```

VMOVAPS __m128 _mm_mask_load_ps(__m128 a, __mmask8 k, void * s);
VMOVAPS __m128 _mm_maskz_load_ps(__mmask8 k, void * s);
VMOVAPS void _mm_mask_store_ps(void * d, __mmask8 k, __m128 a);
MOVAPS __m256 _mm256_load_ps (float * p);
MOVAPS void _mm256_store_ps(float * p, __m256 a);
MOVAPS __m128 _mm_load_ps (float * p);
MOVAPS void _mm_store_ps(float * p, __m128 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE in Table 2-18, "Type 1 Class Exception Conditions," additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-44, "Type E1 Class Exception Conditions."

MOVD/MOVQ—Move Doubleword/Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 6E /r MOVD mm, r/m32	A	V/V	MMX	Move doubleword from r/m32 to mm.
NP REX.W + 0F 6E /r MOVQ mm, r/m64	A	V/N.E.	MMX	Move quadword from r/m64 to mm.
NP 0F 7E /r MOVD r/m32, mm	B	V/V	MMX	Move doubleword from mm to r/m32.
NP REX.W + 0F 7E /r MOVQ r/m64, mm	B	V/N.E.	MMX	Move quadword from mm to r/m64.
66 0F 6E /r MOVD xmm, r/m32	A	V/V	SSE2	Move doubleword from r/m32 to xmm.
66 REX.W 0F 6E /r MOVQ xmm, r/m64	A	V/N.E.	SSE2	Move quadword from r/m64 to xmm.
66 0F 7E /r MOVD r/m32, xmm	B	V/V	SSE2	Move doubleword from xmm register to r/m32.
66 REX.W 0F 7E /r MOVQ r/m64, xmm	B	V/N.E.	SSE2	Move quadword from xmm register to r/m64.
VEX.128.66.0F.W0 6E / VMOVD xmm1, r32/m32	A	V/V	AVX	Move doubleword from r/m32 to xmm1.
VEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64	A	V/N.E.[1]	AVX	Move quadword from r/m64 to xmm1.
VEX.128.66.0F.W0 7E /r VMOVD r32/m32, xmm1	B	V/V	AVX	Move doubleword from xmm1 register to r/m32.
VEX.128.66.0F.W1 7E /r VMOVQ r64/m64, xmm1	B	V/N.E.[1]	AVX	Move quadword from xmm1 register to r/m64.
EVEX.128.66.0F.W0 6E /r VMOVD xmm1, r32/m32	C	V/V	AVX512F OR AVX10.1[2]	Move doubleword from r/m32 to xmm1.
EVEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64	C	V/N.E.	AVX512F OR AVX10.1[2]	Move quadword from r/m64 to xmm1.
EVEX.128.66.0F.W0 7E /r VMOVD r32/m32, xmm1	D	V/V	AVX512F OR AVX10.1[2]	Move doubleword from xmm1 register to r/m32.
EVEX.128.66.0F.W1 7E /r VMOVQ r64/m64, xmm1	D	V/N.E.[1]	AVX512F OR AVX10.1[2]	Move quadword from xmm1 register to r/m64.

NOTES:

1. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Copies a doubleword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be general-purpose registers, MMX technology registers, XMM registers, or 32-bit memory locations. This instruction can be used to move a doubleword to and from the low doubleword of an MMX technology register and a general-purpose register or a 32-bit memory location, or to and from the low doubleword of an XMM register and a general-purpose register or a 32-bit memory location. The instruction cannot be used to transfer data between MMX technology registers, between XMM registers, between general-purpose registers, or between memory locations.

When the destination operand is an MMX technology register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 64 bits. When the destination operand is an XMM register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 128 bits.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

MOVD/Q with XMM destination:

Moves a dword/qword integer from the source operand and stores it in the low 32/64-bits of the destination XMM register. The upper bits of the destination are zeroed. The source operand can be a 32/64-bit register or 32/64-bit memory location.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. Qword operation requires the use of REX.W=1.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. Qword operation requires the use of VEX.W=1.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. Qword operation requires the use of EVEX.W=1.

MOVD/Q with 32/64 reg/mem destination:

Stores the low dword/qword of the source XMM register to 32/64-bit memory location or general-purpose register. Qword operation requires the use of REX.W=1, VEX.W=1, or EVEX.W=1.

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

If VMOVD or VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVD (When Destination Operand is an MMX Technology Register)

```
    DEST[31:0] := SRC;
    DEST[63:32] := 00000000H;
```

MOVD (When Destination Operand is an XMM Register)

```
    DEST[31:0] := SRC;
    DEST[127:32] := 000000000000000000000000H;
    DEST[MAXVL-1:128] (Unmodified)
```

MOVD (When Source Operand is an MMX Technology or XMM Register)

```
    DEST := SRC[31:0];
```

VMOVD (VEX-Encoded Version when Destination is an XMM Register)

```
    DEST[31:0] := SRC[31:0]
    DEST[MAXVL-1:32] := 0
```

MOVQ (When Destination Operand is an XMM Register)

```
    DEST[63:0] := SRC[63:0];
    DEST[127:64] := 0000000000000000H;
    DEST[MAXVL-1:128] (Unmodified)
```

MOVQ (When Destination Operand is r/m64)

```
    DEST[63:0] := SRC[63:0];
```

MOVQ (When Source Operand is an XMM Register or r/m64)

```
    DEST := SRC[63:0];
```

VMOVQ (VEX-Encoded Version When Destination is an XMM Register)

```
    DEST[63:0] := SRC[63:0]
    DEST[MAXVL-1:64] := 0
```

VMOVD (EVEX-Encoded Version When Destination is an XMM Register)

```
DEST[31:0] := SRC[31:0]
DEST[MAXVL-1:32] := 0
```

VMOVQ (EVEX-Encoded Version When Destination is an XMM Register)

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVD __m64 _mm_cvtsi32_si64 (int i )
MOVD int _mm_cvtsi64_si32 ( __m64m )
MOVD __m128i _mm_cvtsi32_si128 (int a)
MOVD int _mm_cvtsi128_si32 ( __m128i a)
MOVQ __int64 _mm_cvtsi128_si64(__m128i);
MOVQ __m128i _mm_cvtsi64_si128(__int64);
VMOVD __m128i _mm_cvtsi32_si128( int);
VMOVD int _mm_cvtsi128_si32( __m128i );
VMOVQ __m128i _mm_cvtsi64_si128 (__int64);
VMOVQ __int64 _mm_cvtsi128_si64(__m128i );
VMOVQ __m128i _mm_loadl_epi64( __m128i * s);
VMOVQ void _mm_storel_epi64( __m128i * d, __m128i s);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”

Additionally:

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

MOVDDUP—Replicate Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 12 /r MOVDDUP xmm1, xmm2/m64	A	V/V	SSE3	Move double precision floating-point value from xmm2/m64 and duplicate into xmm1.
VEX.128.F2.0F.WIG 12 /r VMOVDDUP xmm1, xmm2/m64	A	V/V	AVX	Move double precision floating-point value from xmm2/m64 and duplicate into xmm1.
VEX.256.F2.0F.WIG 12 /r VMOVDDUP ymm1, ymm2/m256	A	V/V	AVX	Move even index double precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F2.0F.W1 12 /r VMOVDDUP xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move double precision floating-point value from xmm2/m64 and duplicate each element into xmm1 subject to writemask k1.
EVEX.256.F2.0F.W1 12 /r VMOVDDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move even index double precision floating-point values from ymm2/m256 and duplicate each element into ymm1 subject to writemask k1.
EVEX.512.F2.0F.W1 12 /r VMOVDDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F OR AVX10.1[1]	Move even index double precision floating-point values from zmm2/m512 and duplicate each element into zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

For 256-bit or higher versions: Duplicates even-indexed double precision floating-point values from the source operand (the second operand) and into adjacent pair and store to the destination operand (the first operand).

For 128-bit versions: Duplicates the low double precision floating-point value from the source operand (the second operand) and store to the destination operand (the first operand).

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register are unchanged. The source operand is XMM register or a 64-bit memory location.

VEX.128 and EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. The source operand is XMM register or a 64-bit memory location. The destination is updated conditionally under the writemask for EVEX version.

VEX.256 and EVEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed. The source operand is YMM register or a 256-bit memory location. The destination is updated conditionally under the writemask for EVEX version.

EVEX.512 encoded version: The destination is updated according to the writemask. The source operand is ZMM register or a 512-bit memory location.

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

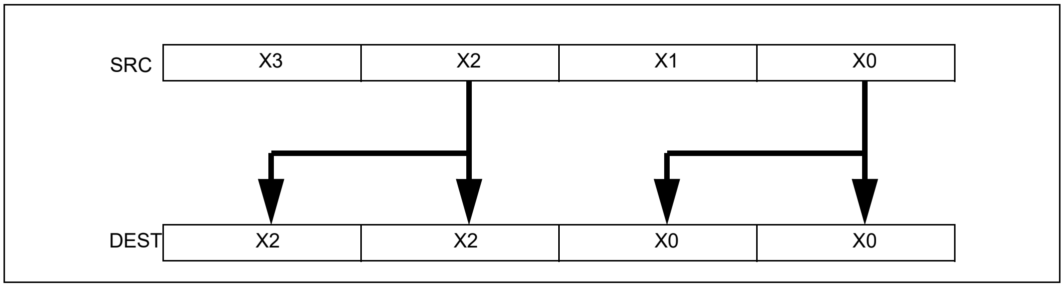


Figure 1-6. VMOVDDUP Operation

Operation

VMOVDDUP (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_SRC[63:0] := SRC[63:0]
TMP_SRC[127:64] := SRC[63:0]
IF VL >= 256
    TMP_SRC[191:128] := SRC[191:128]
    TMP_SRC[255:192] := SRC[191:128]
FI;
IF VL >= 512
    TMP_SRC[319:256] := SRC[319:256]
    TMP_SRC[383:320] := SRC[319:256]
    TMP_SRC[477:384] := SRC[477:384]
    TMP_SRC[511:484] := SRC[477:384]
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_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           ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDDUP (VEX.256 Encoded Version)

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

VMOVDDUP (VEX.128 Encoded Version)

```
DEST[63:0] := SRC[63:0]
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] := 0
```

MOVDDUP (128-bit Legacy SSE Version)

```
DEST[63:0] := SRC[63:0]
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVDDUP __m512d _mm512_movedup_pd( __m512d a);
VMOVDDUP __m512d _mm512_mask_movedup_pd(__m512d s, __mmask8 k, __m512d a);
VMOVDDUP __m512d _mm512_maskz_movedup_pd( __mmask8 k, __m512d a);
VMOVDDUP __m256d _mm256_mask_movedup_pd(__m256d s, __mmask8 k, __m256d a);
VMOVDDUP __m256d _mm256_maskz_movedup_pd( __mmask8 k, __m256d a);
VMOVDDUP __m128d _mm_mask_movedup_pd(__m128d s, __mmask8 k, __m128d a);
VMOVDDUP __m128d _mm_maskz_movedup_pd( __mmask8 k, __m128d a);
MOVDDUP __m256d _mm256_movedup_pd (__m256d a);
MOVDDUP __m128d _mm_movedup_pd (__m128d a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-52, “Type E5NF Class Exception Conditions.”

Additionally:

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

MOVDQA,VMOVDQA32/64—Move Aligned Packed Integer Values

Opcode/ Instruction	**Op/En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
66 0F 6F /r MOVDQA xmm1, xmm2/m128	A	V/V	SSE2	Move aligned packed integer values from xmm2/mem to xmm1.
66 0F 7F /r MOVDQA xmm2/m128, xmm1	B	V/V	SSE2	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 6F /r VMOVDQA xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed integer values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 7F /r VMOVDQA xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 6F /r VMOVDQA ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed integer values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 7F /r VMOVDQA ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed integer values from ymm1 to ymm2/mem.
EVEX.128.66.0F.W0 6F /r VMOVDQA32 xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W0 6F /r VMOVDQA32 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W0 6F /r VMOVDQA32 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1[1]	Move aligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W0 7F /r VMOVDQA32 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W0 7F /r VMOVDQA32 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W0 7F /r VMOVDQA32 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1[1]	Move aligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.66.0F.W1 6F /r VMOVDQA64 xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 6F /r VMOVDQA64 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 6F /r VMOVDQA64 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1[1]	Move aligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 7F /r VMOVDQA64 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 7F /r VMOVDQA64 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move aligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 7F /r VMOVDQA64 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1[1]	Move aligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

EVEX encoded versions:

Moves 128, 256 or 512 bits of packed doubleword/quadword integer values from the source operand (the second operand) to the destination operand (the first operand). This instruction can be used to load a vector register from an int32/int64 memory location, to store the contents of a vector register into an int32/int64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16 (EVEX.128)/32(EVEX.256)/64(EVEX.512)-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

The destination operand is updated at 32-bit (VMOVDQA32) or 64-bit (VMOVDQA64) granularity according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVDQA32 (EVEX Encoded Versions, Register-Copy Form)

```
(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[i+31:i]
        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
```

VMOVDQA32 (EVEX Encoded Versions, Store-Form)

```
(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[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR;
```

VMOVDQA32 (EVEX Encoded Versions, Load-Form)

```
(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[i+31:i]
        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
```

VMOVDQA64 (EVEX Encoded Versions, Register-Copy Form)

```
(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[i+63:i]
        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
```

VMOVDQA64 (EVEX Encoded Versions, Store-Form)

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

VMOVDQA64 (EVEX Encoded Versions, Load-Form)

```
(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[i+63:i]
        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
```

VMOVDQA (VEX.256 Encoded Version, Load - and Register Copy)

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

VMOVDQA (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVDQA (VEX.128 Encoded Version)

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

VMOVDQA (128-bit Load- and Register-Copy- Form Legacy SSE Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVDQA (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVDQA32 __m512i _mm512_load_epi32(void * sa);
VMOVDQA32 __m512i _mm512_mask_load_epi32(__m512i s, __mmask16 k, void * sa);
VMOVDQA32 __m512i _mm512_maskz_load_epi32(__mmask16 k, void * sa);
VMOVDQA32 void _mm512_store_epi32(void * d, __m512i a);
VMOVDQA32 void _mm512_mask_store_epi32(void * d, __mmask16 k, __m512i a);
VMOVDQA32 __m256i _mm256_mask_load_epi32(__m256i s, __mmask8 k, void * sa);
VMOVDQA32 __m256i _mm256_maskz_load_epi32(__mmask8 k, void * sa);

VMOVDQA32 void _mm256_store_epi32(void * d, __m256i a);
VMOVDQA32 void _mm256_mask_store_epi32(void * d, __mmask8 k, __m256i a);
VMOVDQA32 __m128i _mm_mask_load_epi32(__m128i s, __mmask8 k, void * sa);
VMOVDQA32 __m128i _mm_maskz_load_epi32(__mmask8 k, void * sa);
VMOVDQA32 void _mm_store_epi32(void * d, __m128i a);
VMOVDQA32 void _mm_mask_store_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQA64 __m512i _mm512_load_epi64(void * sa);
VMOVDQA64 __m512i _mm512_mask_load_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQA64 __m512i _mm512_maskz_load_epi64(__mmask8 k, void * sa);
VMOVDQA64 void _mm512_store_epi64(void * d, __m512i a);
VMOVDQA64 void _mm512_mask_store_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQA64 __m256i _mm256_mask_load_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQA64 __m256i _mm256_maskz_load_epi64(__mmask8 k, void * sa);
VMOVDQA64 void _mm256_store_epi64(void * d, __m256i a);
VMOVDQA64 void _mm256_mask_store_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQA64 __m128i _mm_mask_load_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQA64 __m128i _mm_maskz_load_epi64(__mmask8 k, void * sa);
VMOVDQA64 void _mm_store_epi64(void * d, __m128i a);
VMOVDQA64 void _mm_mask_store_epi64(void * d, __mmask8 k, __m128i a);
MOVDQA void __m256i _mm256_load_si256 (__m256i * p);
MOVDQA _mm256_store_si256(_m256i *p, __m256i a);
MOVDQA __m128i _mm_load_si128 (__m128i * p);
MOVDQA void _mm_store_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, “Type 1 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-44, “Type E1 Class Exception Conditions.”

Additionally:

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

MOVDQU,VMOVDQU8/16/32/64—Move Unaligned Packed Integer Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 6F /r MOVDQU xmm1, xmm2/m128	A	V/V	SSE2	Move unaligned packed integer values from xmm2/m128 to xmm1.
F3 0F 7F /r MOVDQU xmm2/m128, xmm1	B	V/V	SSE2	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.128.F3.0F.WIG 6F /r VMOVDQU xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed integer values from xmm2/m128 to xmm1.
VEX.128.F3.0F.WIG 7F /r VMOVDQU xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.256.F3.0F.WIG 6F /r VMOVDQU ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed integer values from ymm2/m256 to ymm1.
VEX.256.F3.0F.WIG 7F /r VMOVDQU ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed integer values from ymm1 to ymm2/m256.
EVEX.128.F2.0F.W0 6F /r VMOVDQU8 xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Move unaligned packed byte integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W0 6F /r VMOVDQU8 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Move unaligned packed byte integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W0 6F /r VMOVDQU8 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512BW OR AVX10.1[1]	Move unaligned packed byte integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W0 7F /r VMOVDQU8 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Move unaligned packed byte integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W0 7F /r VMOVDQU8 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Move unaligned packed byte integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W0 7F /r VMOVDQU8 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512BW OR AVX10.1[1]	Move unaligned packed byte integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F2.0F.W1 6F /r VMOVDQU16 xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Move unaligned packed word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W1 6F /r VMOVDQU16 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Move unaligned packed word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W1 6F /r VMOVDQU16 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512BW OR AVX10.1[1]	Move unaligned packed word integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W1 7F /r VMOVDQU16 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Move unaligned packed word integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W1 7F /r VMOVDQU16 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Move unaligned packed word integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W1 7F /r VMOVDQU16 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512BW OR AVX10.1[1]	Move unaligned packed word integer values from zmm1 to zmm2/m512 using writemask k1.

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W0 6F /r VMOVDQU32 xmm1 {k1}{z}, xmm2/mm128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F3.0F.W0 6F /r VMOVDQU32 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W0 6F /r VMOVDQU32 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1[1]	Move unaligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W0 7F /r VMOVDQU32 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F3.0F.W0 7F /r VMOVDQU32 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W0 7F /r VMOVDQU32 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1[1]	Move unaligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F3.0F.W1 6F /r VMOVDQU64 xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F3.0F.W1 6F /r VMOVDQU64 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W1 6F /r VMOVDQU64 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1[1]	Move unaligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W1 7F /r VMOVDQU64 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F3.0F.W1 7F /r VMOVDQU64 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W1 7F /r VMOVDQU64 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1[1]	Move unaligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

EVEX encoded versions:

Moves 128, 256 or 512 bits of packed byte/word/doubleword/quadword integer values from the source operand (the second operand) to the destination operand (first operand). This instruction can be used to load a vector register from a memory location, to store the contents of a vector register into a memory location, or to move data between two vector registers.

The destination operand is updated at 8-bit (VMOVDQU8), 16-bit (VMOVDQU16), 32-bit (VMOVDQU32), or 64-bit (VMOVDQU64) granularity according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned to any alignment without causing a general-protection exception (#GP) to be generated

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVDQU8 (EVEX Encoded Versions, Register-Copy Form)

```
(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[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE  DEST[i+7:i] := 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU8 (EVEX Encoded Versions, Store-Form)

```
(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[i+7:i]
        ELSE *DEST[i+7:i] remains unchanged*            ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU8 (EVEX Encoded Versions, Load-Form)

```
(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[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE  DEST[i+7:i] := 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU16 (EVEX Encoded Versions, Register-Copy Form)

```
(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[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE  DEST[i+15:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU16 (EVEX Encoded Versions, Store-Form)

```
(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[i+15:i]
        ELSE *DEST[i+15:i] remains unchanged*           ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU16 (EVEX Encoded Versions, Load-Form)

```
(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[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE  DEST[i+15:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU32 (EVEX Encoded Versions, Register-Copy Form)

```
(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[i+31:i]
        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
```

VMOVDQU32 (EVEX Encoded Versions, Store-Form)

```
(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[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU32 (EVEX Encoded Versions, Load-Form)

```
(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[i+31:i]
        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
```

VMOVDQU64 (EVEX Encoded Versions, Register-Copy Form)

```
(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[i+63:i]
        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
```

VMOVDQU64 (EVEX Encoded Versions, Store-Form)

```
(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[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*       ; merging-masking

    FI;
ENDFOR;
```

VMOVDQU64 (EVEX Encoded Versions, Load-Form)

```
(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[i+63:i]
        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
```

VMOVDQU (VEX.256 Encoded Version, Load - and Register Copy)

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

VMOVDQU (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVDQU (VEX.128 encoded version)

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

VMOVDQU (128-bit Load- and Register-Copy- Form Legacy SSE Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVDQU (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVDQU16 __m512i _mm512_mask_loadu_epi16(__m512i s, __mmask32 k, void * sa);
VMOVDQU16 __m512i _mm512_maskz_loadu_epi16( __mmask32 k, void * sa);
VMOVDQU16 void _mm512_mask_storeu_epi16(void * d, __mmask32 k, __m512i a);
VMOVDQU16 __m256i _mm256_mask_loadu_epi16(__m256i s, __mmask16 k, void * sa);
VMOVDQU16 __m256i _mm256_maskz_loadu_epi16( __mmask16 k, void * sa);
VMOVDQU16 void _mm256_mask_storeu_epi16(void * d, __mmask16 k, __m256i a);
VMOVDQU16 __m128i _mm_mask_loadu_epi16(__m128i s, __mmask8 k, void * sa);
VMOVDQU16 __m128i _mm_maskz_loadu_epi16( __mmask8 k, void * sa);
VMOVDQU16 void _mm_mask_storeu_epi16(void * d, __mmask8 k, __m128i a);
VMOVDQU32 __m512i _mm512_loadu_epi32( void * sa);
VMOVDQU32 __m512i _mm512_mask_loadu_epi32(__m512i s, __mmask16 k, void * sa);
VMOVDQU32 __m512i _mm512_maskz_loadu_epi32( __mmask16 k, void * sa);
VMOVDQU32 void _mm512_storeu_epi32(void * d, __m512i a);
VMOVDQU32 void _mm512_mask_storeu_epi32(void * d, __mmask16 k, __m512i a);
VMOVDQU32 __m256i _mm256_mask_loadu_epi32(__m256i s, __mmask8 k, void * sa);
VMOVDQU32 __m256i _mm256_maskz_loadu_epi32( __mmask8 k, void * sa);
VMOVDQU32 void _mm256_storeu_epi32(void * d, __m256i a);
VMOVDQU32 void _mm256_mask_storeu_epi32(void * d, __mmask8 k, __m256i a);
VMOVDQU32 __m128i _mm_mask_loadu_epi32(__m128i s, __mmask8 k, void * sa);
VMOVDQU32 __m128i _mm_maskz_loadu_epi32( __mmask8 k, void * sa);
VMOVDQU32 void _mm_storeu_epi32(void * d, __m128i a);
VMOVDQU32 void _mm_mask_storeu_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQU64 __m512i _mm512_loadu_epi64( void * sa);
VMOVDQU64 __m512i _mm512_mask_loadu_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQU64 __m512i _mm512_maskz_loadu_epi64( __mmask8 k, void * sa);
VMOVDQU64 void _mm512_storeu_epi64(void * d, __m512i a);
VMOVDQU64 void _mm512_mask_storeu_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQU64 __m256i _mm256_mask_loadu_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQU64 __m256i _mm256_maskz_loadu_epi64( __mmask8 k, void * sa);
VMOVDQU64 void _mm256_storeu_epi64(void * d, __m256i a);
VMOVDQU64 void _mm256_mask_storeu_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQU64 __m128i _mm_mask_loadu_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQU64 __m128i _mm_maskz_loadu_epi64( __mmask8 k, void * sa);
VMOVDQU64 void _mm_storeu_epi64(void * d, __m128i a);
VMOVDQU64 void _mm_mask_storeu_epi64(void * d, __mmask8 k, __m128i a);
VMOVDQU8 __m512i _mm512_mask_loadu_epi8(__m512i s, __mmask64 k, void * sa);
VMOVDQU8 __m512i _mm512_maskz_loadu_epi8( __mmask64 k, void * sa);
VMOVDQU8 void _mm512_mask_storeu_epi8(void * d, __mmask64 k, __m512i a);
VMOVDQU8 __m256i _mm256_mask_loadu_epi8(__m256i s, __mmask32 k, void * sa);
VMOVDQU8 __m256i _mm256_maskz_loadu_epi8( __mmask32 k, void * sa);
VMOVDQU8 void _mm256_mask_storeu_epi8(void * d, __mmask32 k, __m256i a);
VMOVDQU8 __m128i _mm_mask_loadu_epi8(__m128i s, __mmask16 k, void * sa);
VMOVDQU8 __m128i _mm_maskz_loadu_epi8( __mmask16 k, void * sa);
VMOVDQU8 void _mm_mask_storeu_epi8(void * d, __mmask16 k, __m128i a);
MOVDQU __m256i _mm256_loadu_si256 (__m256i * p);
```

MOVDQU _mm256_storeu_si256(_m256i *p, __m256i a);
MOVDQU __m128i _mm_loadu_si128 (__m128i * p);
MOVDQU _mm_storeu_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

Additionally:

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

MOVHLPS—Move Packed Single Precision Floating-Point Values High to Low

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 12 /r MOVHLPS xmm1, xmm2	RM	V/V	SSE	Move two packed single precision floating-point values from high quadword of xmm2 to low quadword of xmm1.
VEX.128.0F.WIG 12 /r VMOVHLPS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single precision floating-point values from high quadword of xmm3 and low quadword of xmm2.
EVEX.128.0F.W0 12 /r VMOVHLPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F OR AVX10.1[1]	Merge two packed single precision floating-point values from high quadword of xmm3 and low quadword of xmm2.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
RVM	ModRM:reg (w)	VEX.vvvv (r) / EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single precision floating-point values from the high quadword of the second XMM argument (second operand) to the low quadword of the first XMM register (first argument). The quadword at bits 127:64 of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

128-bit and EVEX three-argument form:

Moves two packed single precision floating-point values from the high quadword of the third XMM argument (third operand) to the low quadword of the destination (first operand). Copies the high quadword from the second XMM argument (second operand) to the high quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVHLPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVHLPS (128-bit Two-Argument Form)

```
DEST[63:0] := SRC[127:64]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVHLPS (128-bit Three-Argument Form - VEX & EVEX)

```
DEST[63:0] := SRC2[127:64]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

---

[1] ModRM.MOD = 011B required.

Intel C/C++ Compiler Intrinsic Equivalent

MOVHLPS __m128 _mm_movehl_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions,” additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128 in Table 2-55, “Type E7NM Class Exception Conditions.”

MOVHPD—Move High Packed Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 16 /r MOVHPD xmm1, m64	A	V/V	SSE2	Move double precision floating-point value from m64 to high quadword of xmm1.
VEX.128.66.0F.WIG 16 /r VMOVHPD xmm2, xmm1, m64	B	V/V	AVX	Merge double precision floating-point value from m64 and the low quadword of xmm1.
EVEX.128.66.0F.W1 16 /r VMOVHPD xmm2, xmm1, m64	D	V/V	AVX512F OR AVX10.1[1]	Merge double precision floating-point value from m64 and the low quadword of xmm1.
66 0F 17 /r MOVHPD m64, xmm1	C	V/V	SSE2	Move double precision floating-point value from high quadword of xmm1 to m64.
VEX.128.66.0F.WIG 17 /r VMOVHPD m64, xmm1	C	V/V	AVX	Move double precision floating-point value from high quadword of xmm1 to m64.
EVEX.128.66.0F.W1 17 /r VMOVHPD m64, xmm1	E	V/V	AVX512F OR AVX10.1[1]	Move double precision floating-point value from high quadword of xmm1 to m64.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
E	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double precision floating-point value from the source 64-bit memory operand and stores it in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads a double precision floating-point value from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (second operand) are copied to the low 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores a double precision floating-point value from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPD (store) (VEX.128.66.0F 17 /r) is legal and has the same behavior as the existing 66 0F 17 store. For VMOVHPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPD is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVHPD (128-bit Legacy SSE Load)

```
DEST[63:0] (Unmodified)
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

VMOVHPD (VEX.128 & EVEX Encoded Load)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0
```

VMOVHPD (Store)

```
DEST[63:0] := SRC[127:64]
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPD __m128d _mm_loadh_pd (__m128d a, double *p)
MOVHPD void _mm_storeh_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”

MOVHPS—Move High Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 16 /r MOVHPS xmm1, m64	A	V/V	SSE	Move two packed single precision floating-point values from m64 to high quadword of xmm1.
VEX.128.0F.WIG 16 /r VMOVHPS xmm2, xmm1, m64	B	V/V	AVX	Merge two packed single precision floating-point values from m64 and the low quadword of xmm1.
EVEX.128.0F.W0 16 /r VMOVHPS xmm2, xmm1, m64	D	V/V	AVX512F OR AVX10.1[1]	Merge two packed single precision floating-point values from m64 and the low quadword of xmm1.
NP 0F 17 /r MOVHPS m64, xmm1	C	V/V	SSE	Move two packed single precision floating-point values from high quadword of xmm1 to m64.
VEX.128.0F.WIG 17 /r VMOVHPS m64, xmm1	C	V/V	AVX	Move two packed single precision floating-point values from high quadword of xmm1 to m64.
EVEX.128.0F.W0 17 /r VMOVHPS m64, xmm1	E	V/V	AVX512F OR AVX10.1[1]	Move two packed single precision floating-point values from high quadword of xmm1 to m64.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	Tuple2	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
E	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single precision floating-point values from the source 64-bit memory operand and stores them in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads two single precision floating-point values from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (the second operand) are copied to the lower 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores two packed single precision floating-point values from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPS (store) (VEX.128.0F 17 /r) is legal and has the same behavior as the existing 0F 17 store. For VMOVHPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVHPS (128-bit Legacy SSE Load)

```
DEST[63:0] (Unmodified)
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

VMOVHPS (VEX.128 and EVEX Encoded Load)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0
```

VMOVHPS (Store)

```
DEST[63:0] := SRC[127:64]
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPS __m128 _mm_loadh_pi (__m128 a, __m64 *p)
MOVHPS void _mm_storeh_pi (__m64 *p, __m128 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”

MOVLHPS—Move Packed Single Precision Floating-Point Values Low to High

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 16 /r MOVLHPS xmm1, xmm2	RM	V/V	SSE	Move two packed single precision floating-point values from low quadword of xmm2 to high quadword of xmm1.
VEX.128.0F.WIG 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single precision floating-point values from low quadword of xmm3 and low quadword of xmm2.
EVEX.128.0F.W0 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F OR AVX10.1[1]	Merge two packed single precision floating-point values from low quadword of xmm3 and low quadword of xmm2.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
RVM	ModRM:reg (w)	VEX.vvvv (r) / EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single precision floating-point values from the low quadword of the second XMM argument (second operand) to the high quadword of the first XMM register (first argument). The low quadword of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register are unmodified.

128-bit three-argument forms:

Moves two packed single precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVLHPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVLHPS (128-bit Two-Argument Form)

```
DEST[63:0] (Unmodified)
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

VMOVLHPS (128-bit Three-Argument Form - VEX & EVEX)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0
```

---

[1] ModRM.MOD = 011B required

Intel C/C++ Compiler Intrinsic Equivalent

MOVLHPS __m128 _mm_movelh_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions,” additionally:

#UD If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128 in Table 2-55, “Type E7NM Class Exception Conditions.”

MOVLPD—Move Low Packed Double Precision Floating-Point Value

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
66 0F 12 /r MOVLPD xmm1, m64	A	V/V	SSE2	Move double precision floating-point value from m64 to low quadword of xmm1.
VEX.128.66.0F.WIG 12 /r VMOVLPD xmm2, xmm1, m64	B	V/V	AVX	Merge double precision floating-point value from m64 and the high quadword of xmm1.
EVEX.128.66.0F.W1 12 /r VMOVLPD xmm2, xmm1, m64	D	V/V	AVX512F OR AVX10.1[1]	Merge double precision floating-point value from m64 and the high quadword of xmm1.
66 0F 13/r MOVLPD m64, xmm1	C	V/V	SSE2	Move double precision floating-point value from low quadword of xmm1 to m64.
VEX.128.66.0F.WIG 13/r VMOVLPD m64, xmm1	C	V/V	AVX	Move double precision floating-point value from low quadword of xmm1 to m64.
EVEX.128.66.0F.W1 13/r VMOVLPD m64, xmm1	E	V/V	AVX512F OR AVX10.1[1]	Move double precision floating-point value from low quadword of xmm1 to m64.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (r)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
E	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double precision floating-point value from the source 64-bit memory operand and stores it in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads a double precision floating-point value from the source 64-bit memory operand (third operand), merges it with the upper 64-bits of the first source XMM register (second operand), and stores it in the low 128-bits of the destination XMM register (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores a double precision floating-point value from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPD (store) (VEX.128.66.0F 13 /r) is legal and has the same behavior as the existing 66 0F 13 store. For VMOVLPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPD is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVLPD (128-bit Legacy SSE Load)

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVLPD (VEX.128 & EVEX Encoded Load)

```
DEST[63:0] := SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VMOVLPD (Store)

```
DEST[63:0] := SRC[63:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPD __m128d _mm_loadl_pd (__m128d a, double *p)

MOVLPD void _mm_storel_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”

MOVLPS—Move Low Packed Single Precision Floating-Point Values

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 12 /r MOVLPS xmm1, m64	A	V/V	SSE	Move two packed single precision floating-point values from m64 to low quadword of xmm1.
VEX.128.0F.WIG 12 /r VMOVLPS xmm2, xmm1, m64	B	V/V	AVX	Merge two packed single precision floating-point values from m64 and the high quadword of xmm1.
EVEX.128.0F.W0 12 /r VMOVLPS xmm2, xmm1, m64	D	V/V	AVX512F OR AVX10.1[1]	Merge two packed single precision floating-point values from m64 and the high quadword of xmm1.
0F 13/r MOVLPS m64, xmm1	C	V/V	SSE	Move two packed single precision floating-point values from low quadword of xmm1 to m64.
VEX.128.0F.WIG 13/r VMOVLPS m64, xmm1	C	V/V	AVX	Move two packed single precision floating-point values from low quadword of xmm1 to m64.
EVEX.128.0F.W0 13/r VMOVLPS m64, xmm1	E	V/V	AVX512F OR AVX10.1[1]	Move two packed single precision floating-point values from low quadword of xmm1 to m64.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	Tuple2	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
E	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single precision floating-point values from the source 64-bit memory operand and stores them in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads two packed single precision floating-point values from the source 64-bit memory operand (the third operand), merges them with the upper 64-bits of the first source operand (the second operand), and stores them in the low 128-bits of the destination register (the first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Loads two packed single precision floating-point values from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPS (store) (VEX.128.0F 13 /r) is legal and has the same behavior as the existing 0F 13 store. For VMOVLPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVLPS (128-bit Legacy SSE Load)

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVLPS (VEX.128 & EVEX Encoded Load)

```
DEST[63:0] := SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VMOVLPS (Store)

```
DEST[63:0] := SRC[63:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVLPS __m128 _mm_loadl_pi ( __m128 a, __m64 *p)
MOVLPS void _mm_storel_pi (__m64 *p, __m128 a)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:

#UD If VEX.L = 1.

EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”

MOVNTDQA—Load Double Quadword Non-Temporal Aligned Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2A /r MOVNTDQA xmm1, m128	A	V/V	SSE4_1	Move double quadword from m128 to xmm1 using non-temporal hint if WC memory type.
VEX.128.66.0F38.WIG 2A /r VMOVNTDQA xmm1, m128	A	V/V	AVX	Move double quadword from m128 to xmm using non-temporal hint if WC memory type.
VEX.256.66.0F38.WIG 2A /r VMOVNTDQA ymm1, m256	A	V/V	AVX2	Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.
EVEX.128.66.0F38.W0 2A /r VMOVNTDQA xmm1, m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move 128-bit data from m128 to xmm using non-temporal hint if WC memory type.
EVEX.256.66.0F38.W0 2A /r VMOVNTDQA ymm1, m256	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.
EVEX.512.66.0F38.W0 2A /r VMOVNTDQA zmm1, m512	B	V/V	AVX512F OR AVX10.1[1]	Move 512-bit data from m512 to zmm using non-temporal hint if WC memory type.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding[1]

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

Description

MOVNTDQA loads a double quadword from the source operand (second operand) to the destination operand (first operand) using a non-temporal hint if the memory source is WC (write combining) memory type. For WC memory type, the nontemporal hint may be implemented by loading a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Any memory-type aliased lines in the cache will be snooped and flushed. Subsequent MOVNTDQA reads to unread portions of the WC cache line will receive data from the temporary internal buffer if data is available. The temporary internal buffer may be flushed by the processor at any time for any reason, for example:

- A load operation other than a MOVNTDQA which references memory already resident in a temporary internal buffer.
- A non-WC reference to memory already resident in a temporary internal buffer.
- Interleaving of reads and writes to a single temporary internal buffer.
- Repeated (V)MOVNTDQA loads of a particular 16-byte item in a streaming line.
- Certain micro-architectural conditions including resource shortages, detection of

a mis-speculation condition, and various fault conditions

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when reading the data from memory. Using this protocol, the processor does not read the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being read can override the non-temporal hint, if the memory address specified for the non-temporal read is not a WC

---

[1] ModRM.MOD != 011B

memory region. Information on non-temporal reads and writes can be found in “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the Intel® 64 and IA-32 Architecture Software Developer’s Manual, Volume 3A.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with a MFENCE instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might use different memory types for the referenced memory locations or to synchronize reads of a processor with writes by other agents in the system. A processor’s implementation of the streaming load hint does not override the effective memory type, but the implementation of the hint is processor dependent. For example, a processor implementation may choose to ignore the hint and process the instruction as a normal MOVDQA for any memory type. Alternatively, another implementation may optimize cache reads generated by MOVNTDQA on WB memory type to reduce cache evictions.

The 128-bit (V)MOVNTDQA addresses must be 16-byte aligned or the instruction will cause a #GP.

The 256-bit VMOVNTDQA addresses must be 32-byte aligned or the instruction will cause a #GP.

The 512-bit VMOVNTDQA addresses must be 64-byte aligned or the instruction will cause a #GP.

Operation

MOVNTDQA (128bit- Legacy SSE Form)

```
DEST := SRC
DEST[MAXVL-1:128] (Unmodified)
```

VMOVNTDQA (VEX.128 and EVEX.128 Encoded Form)

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

VMOVNTDQA (VEX.256 and EVEX.256 Encoded Forms)

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

VMOVNTDQA (EVEX.512 Encoded Form)

```
DEST[511:0] := SRC[511:0]
DEST[MAXVL-1:512] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVNTDQA __m512i _mm512_stream_load_si512(__m512i const* p);
MOVNTDQA __m128i _mm_stream_load_si128 (const __m128i *p);
VMOVNTDQA __m256i _mm256_stream_load_si256 (__m256i const* p);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-18, “Type 1 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-45, “Type E1NF Class Exception Conditions.”

Additionally:

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

MOVNTDQ—Store Packed Integers Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E7 /r MOVNTDQ m128, xmm1	A	V/V	SSE2	Move packed integer values in xmm1 to m128 using non-temporal hint.
VEX.128.66.0F.WIG E7 /r VMOVNTDQ m128, xmm1	A	V/V	AVX	Move packed integer values in xmm1 to m128 using non-temporal hint.
VEX.256.66.0F.WIG E7 /r VMOVNTDQ m256, ymm1	A	V/V	AVX	Move packed integer values in ymm1 to m256 using non-temporal hint.
EVEX.128.66.0F.W0 E7 /r VMOVNTDQ m128, xmm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move packed integer values in xmm1 to m128 using non-temporal hint.
EVEX.256.66.0F.W0 E7 /r VMOVNTDQ m256, ymm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move packed integer values in zmm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W0 E7 /r VMOVNTDQ m512, zmm1	B	V/V	AVX512F OR AVX10.1[1]	Move packed integer values in zmm1 to m512 using non-temporal hint.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding[1]

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

Description

Moves the packed integers in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain integer data (packed bytes, words, doublewords, or quadwords). The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (512-bit version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with VMOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

---

[1] ModRM.MOD != 011B

Operation

VMOVNTDQ(EVEX Encoded Versions)

```
VL = 128, 256, 512
DEST[VL-1:0] := SRC[VL-1:0]
DEST[MAXVL-1:VL] := 0
```

MOVNTDQ (Legacy and VEX Versions)

```
DEST := SRC
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTDQ void _mm512_stream_si512(void * p, __m512i a);
VMOVNTDQ void _mm256_stream_si256 (__m256i * p, __m256i a);
MOVNTDQ void _mm_stream_si128 (__m128i * p, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, “Type 1 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-45, “Type E1NF Class Exception Conditions.”

Additionally:

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

MOVNTPD—Store Packed Double Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2B /r MOVNTPD m128, xmm1	A	V/V	SSE2	Move packed double precision values in xmm1 to m128 using non-temporal hint.
VEX.128.66.0F.WIG 2B /r VMOVNTPD m128, xmm1	A	V/V	AVX	Move packed double precision values in xmm1 to m128 using non-temporal hint.
VEX.256.66.0F.WIG 2B /r VMOVNTPD m256, ymm1	A	V/V	AVX	Move packed double precision values in ymm1 to m256 using non-temporal hint.
EVEX.128.66.0F.W1 2B /r VMOVNTPD m128, xmm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move packed double precision values in xmm1 to m128 using non-temporal hint.
EVEX.256.66.0F.W1 2B /r VMOVNTPD m256, ymm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move packed double precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W1 2B /r VMOVNTPD m512, zmm1	B	V/V	AVX512F OR AVX10.1[1]	Move packed double precision values in zmm1 to m512 using non-temporal hint.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding[1]

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

Description

Moves the packed double precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed double precision, floating-pointing data. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

---

[1] ModRM.MOD != 011B

Operation

VMOVNTPD (EVEX Encoded Versions)

```
VL = 128, 256, 512
DEST[VL-1:0] := SRC[VL-1:0]
DEST[MAXVL-1:VL] := 0
```

MOVNTPD (Legacy and VEX Versions)

```
DEST := SRC
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTPD void _mm512_stream_pd(double * p, __m512d a);
VMOVNTPD void _mm256_stream_pd (double * p, __m256d a);
MOVNTPD void _mm_stream_pd (double * p, __m128d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, “Type 1 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-45, “Type E1NF Class Exception Conditions.”

Additionally:

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

MOVNTPS—Store Packed Single Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2B /r MOVNTPS m128, xmm1	A	V/V	SSE	Move packed single precision values xmm1 to mem using non-temporal hint.
VEX.128.0F.WIG 2B /r VMOVNTPS m128, xmm1	A	V/V	AVX	Move packed single precision values xmm1 to mem using non-temporal hint.
VEX.256.0F.WIG 2B /r VMOVNTPS m256, ymm1	A	V/V	AVX	Move packed single precision values ymm1 to mem using non-temporal hint.
EVEX.128.0F.W0 2B /r VMOVNTPS m128, xmm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move packed single precision values in xmm1 to m128 using non-temporal hint.
EVEX.256.0F.W0 2B /r VMOVNTPS m256, ymm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move packed single precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.0F.W0 2B /r VMOVNTPS m512, zmm1	B	V/V	AVX512F OR AVX10.1[1]	Move packed single precision values in zmm1 to m512 using non-temporal hint.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding[1]

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

Description

Moves the packed single precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed single precision, floating-pointing. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

---

[1] ModRM.MOD != 011B

Operation

VMOVNTPS (EVEX Encoded Versions)

```
VL = 128, 256, 512
DEST[VL-1:0] := SRC[VL-1:0]
DEST[MAXVL-1:VL] := 0
```

MOVNTPS

```
DEST := SRC
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTPS void _mm512_stream_ps(float * p, __m512d a);
MOVNTPS void _mm_stream_ps (float * p, __m128d a);
VMOVNTPS void _mm256_stream_ps (float * p, __m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE in Table 2-18, “Type 1 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-45, “Type E1NF Class Exception Conditions.”
Additionally:

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

MOVQ—Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 6F /r MOVQ mm, mm/m64	A	V/V	MMX	Move quadword from mm/m64 to mm.
NP 0F 7F /r MOVQ mm/m64, mm	B	V/V	MMX	Move quadword from mm to mm/m64.
F3 0F 7E /r MOVQ xmm1, xmm2/m64	A	V/V	SSE2	Move quadword from xmm2/mem64 to xmm1.
VEX.128.F3.0F.WIG 7E /r VMOVQ xmm1, xmm2/m64	A	V/V	AVX	Move quadword from xmm2 to xmm1.
EVEX.128.F3.0F.W1 7E /r VMOVQ xmm1, xmm2/m64	C	V/V	AVX512F OR AVX10.1[1]	Move quadword from xmm2/m64 to xmm1.
66 0F D6 /r MOVQ xmm2/m64, xmm1	B	V/V	SSE2	Move quadword from xmm1 to xmm2/mem64.
VEX.128.66.0F.WIG D6 /r VMOVQ xmm1/m64, xmm2	B	V/V	AVX	Move quadword from xmm2 register to xmm1/m64.
EVEX.128.66.0F.W1 D6 /r VMOVQ xmm1/m64, xmm2	D	V/V	AVX512F OR AVX10.1[1]	Move quadword from xmm2 register to xmm1/m64.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Copies a quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be MMX technology registers, XMM registers, or 64-bit memory locations. This instruction can be used to move a quadword between two MMX technology registers or between an MMX technology register and a 64-bit memory location, or to move data between two XMM registers or between an XMM register and a 64-bit memory location. The instruction cannot be used to transfer data between memory locations.

When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared to all 0s.

In 64-bit mode and if not encoded using VEX/EVEX, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

If VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVQ Instruction When Operating on MMX Technology Registers and Memory Locations

```
    DEST := SRC;
```

MOVQ Instruction When Source and Destination Operands are XMM Registers

```
    DEST[63:0] := SRC[63:0];
    DEST[127:64] := 0000000000000000H;
```

MOVQ Instruction When Source Operand is XMM Register and Destination

operand is memory location:

```
    DEST := SRC[63:0];
```

MOVQ Instruction When Source Operand is Memory Location and Destination

operand is XMM register:

```
    DEST[63:0] := SRC;
    DEST[127:64] := 0000000000000000H;
```

VMOVQ (VEX.128.F3.0F 7E) With XMM Register Source and Destination

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

VMOVQ (VEX.128.66.0F D6) With XMM Register Source and Destination

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

VMOVQ (7E - EVEX Encoded Version) With XMM Register Source and Destination

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

VMOVQ (D6 - EVEX Encoded Version) With XMM Register Source and Destination

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

VMOVQ (7E) With Memory Source

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

VMOVQ (7E - EVEX Encoded Version) With Memory Source

```
DEST[63:0] := SRC[63:0]
DEST[:MAXVL-1:64] := 0
```

VMOVQ (D6) With Memory DEST

```
DEST[63:0] := SRC2[63:0]
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVQ __m128i _mm_loadu_si64( void * s);
VMOVQ void _mm_storeu_si64( void * d, __m128i s);
MOVQ m128i _mm_move_epi64(__m128i a)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 23-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

MOVSD—Move or Merge Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 10 /r MOVSD xmm1, xmm2	A	V/V	SSE2	Move scalar double precision floating-point value from xmm2 to xmm1 register.
F2 0F 10 /r MOVSD xmm1, m64	A	V/V	SSE2	Load scalar double precision floating-point value from m64 to xmm1 register.
F2 0F 11 /r MOVSD xmm1/m64, xmm2	C	V/V	SSE2	Move scalar double precision floating-point value from xmm2 register to xmm1/m64.
VEX.LIG.F2.0F.WIG 10 /r VMOVSD xmm1, xmm2, xmm3	B	V/V	AVX	Merge scalar double precision floating-point value from xmm2 and xmm3 to xmm1 register.
VEX.LIG.F2.0F.WIG 10 /r VMOVSD xmm1, m64	D	V/V	AVX	Load scalar double precision floating-point value from m64 to xmm1 register.
VEX.LIG.F2.0F.WIG 11 /r VMOVSD xmm1, xmm2, xmm3	E	V/V	AVX	Merge scalar double precision floating-point value from xmm2 and xmm3 registers to xmm1.
VEX.LIG.F2.0F.WIG 11 /r VMOVSD m64, xmm1	C	V/V	AVX	Store scalar double precision floating-point value from xmm1 register to m64.
EVEX.LLIG.F2.0F.W1 10 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	B	V/V	AVX512F OR AVX10.1[1]	Merge scalar double precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LLIG.F2.0F.W1 10 /r VMOVSD xmm1 {k1}{z}, m64	F	V/V	AVX512F OR AVX10.1[1]	Load scalar double precision floating-point value from m64 to xmm1 register under writemask k1.
EVEX.LLIG.F2.0F.W1 11 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	E	V/V	AVX512F OR AVX10.1[1]	Merge scalar double precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LLIG.F2.0F.W1 11 /r VMOVSD m64 {k1}, xmm1	G	V/V	AVX512F OR AVX10.1[1]	Store scalar double precision floating-point value from xmm1 register to m64 under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
E	N/A	ModRM:r/m (w)	EVEX.vvvv (r)	ModRM:reg (r)	N/A
F	Tuple1 Scalar	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
G	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves a scalar double precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 64-bit memory locations. This instruction can be used to move a double precision floating-point value to and from the low quadword of an

XMM register and a 64-bit memory location, or to move a double precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits MAXVL:64 of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the quadword at bits 127:64 of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar double precision floating-point value from the second source operand (the third operand) to the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory store syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:64 of the destination operand is cleared to all 0s.

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

Note: For VMOVSD (memory store and load forms), VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instruction will #UD.

Operation

VMOVSD (EVEX.LLIG.F2.0F 10 /r: VMOVSD xmm1, m64 With Support for 32 Registers)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC[63:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[MAXVL-1:64] := 0
```

VMOVSD (EVEX.LLIG.F2.0F 11 /r: VMOVSD m64, xmm1 With Support for 32 Registers)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC[63:0]
    ELSE    *DEST[63:0] remains unchanged*          ; merging-masking
FI;
```

VMOVSD (EVEX.LLIG.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)

```
IF k1[0] or *no writemask*
    THEN    DEST[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] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

MOVSD (128-bit Legacy SSE Version: MOVSD xmm1, xmm2)

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVSD (VEX.128.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)

```
DEST[63:0] := SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VMOVSD (VEX.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)

```
DEST[63:0] := SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VMOVSD (VEX.128.F2.0F 10 /r: VMOVSD xmm1, m64)

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

MOVSD/VMOVSD (128-bit Versions: MOVSD m64, xmm1 or VMOVSD m64, xmm1)

```
DEST[63:0] := SRC[63:0]
```

MOVSD (128-bit Legacy SSE Version: MOVSD xmm1, m64)

```
DEST[63:0] := SRC[63:0]
DEST[127:64] := 0
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVSD __m128d _mm_mask_load_sd(__m128d s, __mmask8 k, double * p);
VMOVSD __m128d _mm_maskz_load_sd( __mmask8 k, double * p);
VMOVSD __m128d _mm_mask_move_sd(__m128d sh, __mmask8 k, __m128d sl, __m128d a);
VMOVSD __m128d _mm_maskz_move_sd( __mmask8 k, __m128d s, __m128d a);
VMOVSD void _mm_mask_store_sd(double * p, __mmask8 k, __m128d s);
MOVSD __m128d _mm_load_sd (double *p)
MOVSD void _mm_store_sd (double *p, __m128d a)
MOVSD __m128d _mm_move_sd ( __m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:

#UD If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-58, “Type E10 Class Exception Conditions.”

MOVSHDUP—Replicate Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 16 /r MOVSHDUP xmm1, xmm2/m128	A	V/V	SSE3	Move odd index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.128.F3.0F.WIG 16 /r VMOVSHDUP xmm1, xmm2/m128	A	V/V	AVX	Move odd index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.256.F3.0F.WIG 16 /r VMOVSHDUP ymm1, ymm2/m256	A	V/V	AVX	Move odd index single precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F3.0F.W0 16 /r VMOVSHDUP xmm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move odd index single precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.
EVEX.256.F3.0F.W0 16 /r VMOVSHDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move odd index single precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.
EVEX.512.F3.0F.W0 16 /r VMOVSHDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F OR AVX10.1[1]	Move odd index single precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Duplicates odd-indexed single precision floating-point values from the source operand (the second operand) to adjacent element pair in the destination operand (the first operand). See Figure 1-7. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

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

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

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

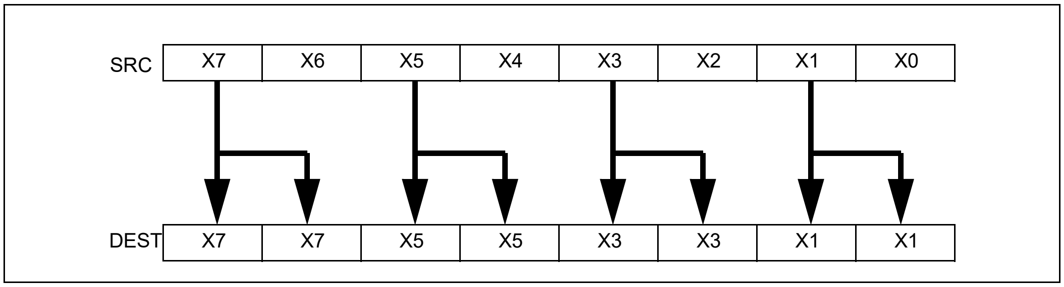


Figure 1-7. MOVSHDUP Operation

Operation

VMOVSHDUP (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
TMP_SRC[31:0] := SRC[63:32]
TMP_SRC[63:32] := SRC[63:32]
TMP_SRC[95:64] := SRC[127:96]
TMP_SRC[127:96] := SRC[127:96]
IF VL >= 256
    TMP_SRC[159:128] := SRC[191:160]
    TMP_SRC[191:160] := SRC[191:160]
    TMP_SRC[223:192] := SRC[255:224]
    TMP_SRC[255:224] := SRC[255:224]
FI;
IF VL >= 512
    TMP_SRC[287:256] := SRC[319:288]
    TMP_SRC[319:288] := SRC[319:288]
    TMP_SRC[351:320] := SRC[383:352]
    TMP_SRC[383:352] := SRC[383:352]
    TMP_SRC[415:384] := SRC[447:416]
    TMP_SRC[447:416] := SRC[447:416]
    TMP_SRC[479:448] := SRC[511:480]
    TMP_SRC[511:480] := SRC[511:480]
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_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
```

VMOVSHDUP (VEX.256 Encoded Version)

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

VMOVSHDUP (VEX.128 Encoded Version)

```
DEST[31:0] := SRC[63:32]
DEST[63:32] := SRC[63:32]
DEST[95:64] := SRC[127:96]
DEST[127:96] := SRC[127:96]
DEST[MAXVL-1:128] := 0
```

MOVSHDUP (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC[63:32]
DEST[63:32] := SRC[63:32]
DEST[95:64] := SRC[127:96]
DEST[127:96] := SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVSHDUP __m512 _mm512_movehdup_ps( __m512 a);
VMOVSHDUP __m512 _mm512_mask_movehdup_ps(__m512 s, __mmask16 k, __m512 a);
VMOVSHDUP __m512 _mm512_maskz_movehdup_ps( __mmask16 k, __m512 a);
VMOVSHDUP __m256 _mm256_mask_movehdup_ps(__m256 s, __mmask8 k, __m256 a);
VMOVSHDUP __m256 _mm256_maskz_movehdup_ps( __mmask8 k, __m256 a);
VMOVSHDUP __m128 _mm_mask_movehdup_ps(__m128 s, __mmask8 k, __m128 a);
VMOVSHDUP __m128 _mm_maskz_movehdup_ps( __mmask8 k, __m128 a);
VMOVSHDUP __m256 _mm256_movehdup_ps (__m256 a);
VMOVSHDUP __m128 _mm_movehdup_ps (__m128 a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions."

Additionally:

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

MOVSLDUP—Replicate Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 12 /r MOVSLDUP xmm1, xmm2/m128	A	V/V	SSE3	Move even index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.128.F3.0F.WIG 12 /r VMOVSLDUP xmm1, xmm2/m128	A	V/V	AVX	Move even index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.256.F3.0F.WIG 12 /r VMOVSLDUP ymm1, ymm2/m256	A	V/V	AVX	Move even index single precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F3.0F.W0 12 /r VMOVSLDUP xmm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move even index single precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.
EVEX.256.F3.0F.W0 12 /r VMOVSLDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move even index single precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.
EVEX.512.F3.0F.W0 12 /r VMOVSLDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F OR AVX10.1[1]	Move even index single precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Duplicates even-indexed single precision floating-point values from the source operand (the second operand). See Figure 1-8. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

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

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

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

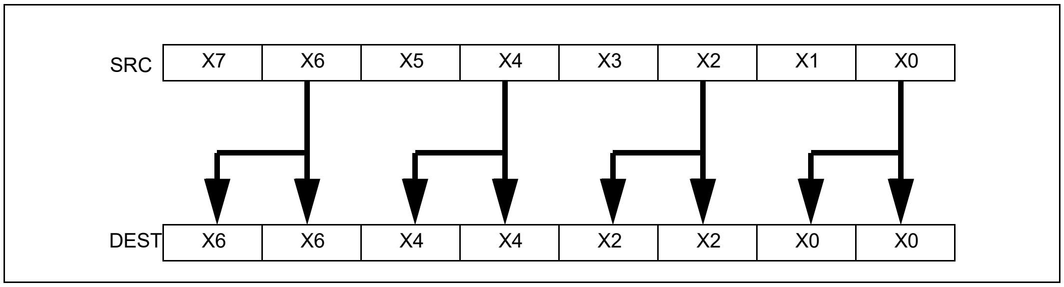


Figure 1-8. MOVSLDUP Operation

Operation

VMOVSLDUP (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
TMP_SRC[31:0] := SRC[31:0]
TMP_SRC[63:32] := SRC[31:0]
TMP_SRC[95:64] := SRC[95:64]
TMP_SRC[127:96] := SRC[95:64]
IF VL >= 256
    TMP_SRC[159:128] := SRC[159:128]
    TMP_SRC[191:160] := SRC[159:128]
    TMP_SRC[223:192] := SRC[223:192]
    TMP_SRC[255:224] := SRC[223:192]
FI;
IF VL >= 512
    TMP_SRC[287:256] := SRC[287:256]
    TMP_SRC[319:288] := SRC[287:256]
    TMP_SRC[351:320] := SRC[351:320]
    TMP_SRC[383:352] := SRC[351:320]
    TMP_SRC[415:384] := SRC[415:384]
    TMP_SRC[447:416] := SRC[415:384]
    TMP_SRC[479:448] := SRC[479:448]
    TMP_SRC[511:480] := SRC[479:448]
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_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
```

VMOVSLDUP (VEX.256 Encoded Version)

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

VMOVSLDUP (VEX.128 Encoded Version)

```
DEST[31:0] := SRC[31:0]
DEST[63:32] := SRC[31:0]
DEST[95:64] := SRC[95:64]
DEST[127:96] := SRC[95:64]
DEST[MAXVL-1:128] := 0
```

MOVSLDUP (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC[31:0]
DEST[63:32] := SRC[31:0]
DEST[95:64] := SRC[95:64]
DEST[127:96] := SRC[95:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVSLDUP __m512 _mm512_moveldup_ps( __m512 a);
VMOVSLDUP __m512 _mm512_mask_moveldup_ps(__m512 s, __mmask16 k, __m512 a);
VMOVSLDUP __m512 _mm512_maskz_moveldup_ps( __mmask16 k, __m512 a);
VMOVSLDUP __m256 _mm256_mask_moveldup_ps(__m256 s, __mmask8 k, __m256 a);
VMOVSLDUP __m256 _mm256_maskz_moveldup_ps( __mmask8 k, __m256 a);
VMOVSLDUP __m128 _mm_mask_moveldup_ps(__m128 s, __mmask8 k, __m128 a);
VMOVSLDUP __m128 _mm_maskz_moveldup_ps( __mmask8 k, __m128 a);
VMOVSLDUP __m256 _mm256_moveldup_ps (__m256 a);
VMOVSLDUP __m128 _mm_moveldup_ps (__m128 a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

Additionally:

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

MOVSS—Move or Merge Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 10 /r MOVSS xmm1, xmm2	A	V/V	SSE	Merge scalar single precision floating-point value from xmm2 to xmm1 register.
F3 0F 10 /r MOVSS xmm1, m32	A	V/V	SSE	Load scalar single precision floating-point value from m32 to xmm1 register.
VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, xmm2, xmm3	B	V/V	AVX	Merge scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register
VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, m32	D	V/V	AVX	Load scalar single precision floating-point value from m32 to xmm1 register.
F3 0F 11 /r MOVSS xmm2/m32, xmm1	C	V/V	SSE	Move scalar single precision floating-point value from xmm1 register to xmm2/m32.
VEX.LIG.F3.0F.WIG 11 /r VMOVSS xmm1, xmm2, xmm3	E	V/V	AVX	Move scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register.
VEX.LIG.F3.0F.WIG 11 /r VMOVSS m32, xmm1	C	V/V	AVX	Move scalar single precision floating-point value from xmm1 register to m32.
EVEX.LLIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	B	V/V	AVX512F OR AVX10.1[1]	Move scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LLIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, m32	F	V/V	AVX512F OR AVX10.1[1]	Move scalar single precision floating-point values from m32 to xmm1 under writemask k1.
EVEX.LLIG.F3.0F.W0 11 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	E	V/V	AVX512F OR AVX10.1[1]	Move scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LLIG.F3.0F.W0 11 /r VMOVSS m32 {k1}, xmm1	G	V/V	AVX512F OR AVX10.1[1]	Move scalar single precision floating-point values from xmm1 to m32 under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
E	N/A	ModRM:r/m (w)	EVEX.vvvv (r)	ModRM:reg (r)	N/A
F	Tuple1 Scalar	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
G	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves a scalar single precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single precision floating-point value to and from the low doubleword of an

XMM register and a 32-bit memory location, or to move a single precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits (MAXVL-1:32) of the corresponding destination register are unmodified. When the source operand is a memory location and destination operand is an XMM registers, Bits (127:32) of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar single precision floating-point value from the second source operand (the third operand) to the low doubleword element of the destination operand (the first operand). Bits 127:32 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory load syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:32 of the destination operand is cleared to all 0s.

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

Note: For memory store form instruction “VMOVSS m32, xmm1”, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD. For memory store form instruction “VMOVSS mv {k1}, xmm1”, EVEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Software should ensure VMOVSS is encoded with VEX.L=0. Encoding VMOVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMOVSS (EVEX.LLIG.F3.0F.W0 11 /r When the Source Operand is Memory and the Destination is an XMM Register)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC[31:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[MAXVL-1:32] := 0
```

VMOVSS (EVEX.LLIG.F3.0F.W0 10 /r When the Source Operand is an XMM Register and the Destination is Memory)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC[31:0]
    ELSE    *DEST[31:0] remains unchanged*           ; merging-masking
FI;
```

VMOVSS (EVEX.LLIG.F3.0F.W0 10/11 /r Where the Source and Destination are XMM Registers)

```
IF k1[0] or *no writemask*
    THEN    DEST[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] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

MOVSS (Legacy SSE Version When the Source and Destination Operands are Both XMM Registers)

```
DEST[31:0] := SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

VMOVSS (VEX.128.F3.0F 11 /r Where the Destination is an XMM Register)

```
DEST[31:0] := SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VMOVSS (VEX.128.F3.0F 10 /r Where the Source and Destination are XMM Registers)

```
DEST[31:0] := SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VMOVSS (VEX.128.F3.0F 10 /r When the Source Operand is Memory and the Destination is an XMM Register)

```
DEST[31:0] := SRC[31:0]
DEST[MAXVL-1:32] := 0
```

MOVSS/VMOVSS (When the Source Operand is an XMM Register and the Destination is Memory)

```
DEST[31:0] := SRC[31:0]
```

MOVSS (Legacy SSE Version when the Source Operand is Memory and the Destination is an XMM Register)

```
DEST[31:0] := SRC[31:0]
DEST[127:32] := 0
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVSS __m128 _mm_mask_load_ss(__m128 s, __mmask8 k, float * p);
VMOVSS __m128 _mm_maskz_load_ss( __mmask8 k, float * p);
VMOVSS __m128 _mm_mask_move_ss(__m128 sh, __mmask8 k, __m128 sl, __m128 a);
VMOVSS __m128 _mm_maskz_move_ss( __mmask8 k, __m128 s, __m128 a);
VMOVSS void _mm_mask_store_ss(float * p, __mmask8 k, __m128 a);
MOVSS __m128 _mm_load_ss(float * p)
MOVSS void_mm_store_ss(float * p, __m128 a)
MOVSS __m128 _mm_move_ss(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:

#UD If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-58, “Type E10 Class Exception Conditions.”

MOVUPD—Move Unaligned Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 10 /r MOVUPD xmm1, xmm2/m128	A	V/V	SSE2	Move unaligned packed double precision floating-point from xmm2/mem to xmm1.
66 0F 11 /r MOVUPD xmm2/m128, xmm1	B	V/V	SSE2	Move unaligned packed double precision floating-point from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 10 /r VMOVUPD xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed double precision floating-point from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 11 /r VMOVUPD xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed double precision floating-point from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 10 /r VMOVUPD ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed double precision floating-point from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 11 /r VMOVUPD ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed double precision floating-point from ymm1 to ymm2/mem.
EVEX.128.66.0F.W1 10 /r VMOVUPD xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed double precision floating-point from xmm2/m128 to xmm1 using writemask k1.
EVEX.128.66.0F.W1 11 /r VMOVUPD xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed double precision floating-point from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 10 /r VMOVUPD ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed double precision floating-point from ymm2/m256 to ymm1 using writemask k1.
EVEX.256.66.0F.W1 11 /r VMOVUPD ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed double precision floating-point from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 10 /r VMOVUPD zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1[1]	Move unaligned packed double precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.512.66.0F.W1 11 /r VMOVUPD zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1[1]	Move unaligned packed double precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

EVEX.512 encoded version:

Moves 512 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a float64 memory location, to store the contents of a ZMM register into a memory. The destination operand is updated according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVUPD (EVEX Encoded Versions, Register-Copy Form)

```
(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[i+63:i]
        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
```

VMOVUPD (EVEX Encoded Versions, Store-Form)

```
(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[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*        ; merging-masking

    FI;
ENDFOR;
```

VMOVUPD (EVEX Encoded Versions, Load-Form)

```
(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[i+63:i]
        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
```

VMOVUPD (VEX.256 Encoded Version, Load - and Register Copy)

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

VMOVUPD (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVUPD (VEX.128 Encoded Version)

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

MOVUPD (128-bit Load- and Register-Copy- Form Legacy SSE Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVUPD (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVUPD __m512d _mm512_loadu_pd( void * s);
VMOVUPD __m512d _mm512_mask_loadu_pd(__m512d a, __mmask8 k, void * s);
VMOVUPD __m512d _mm512_maskz_loadu_pd( __mmask8 k, void * s);
VMOVUPD void _mm512_storeu_pd( void * d, __m512d a);
VMOVUPD void _mm512_mask_storeu_pd( void * d, __mmask8 k, __m512d a);
VMOVUPD __m256d _mm256_mask_loadu_pd(__m256d s, __mmask8 k, void * m);
VMOVUPD __m256d _mm256_maskz_loadu_pd( __mmask8 k, void * m);
VMOVUPD void _mm256_mask_storeu_pd( void * d, __mmask8 k, __m256d a);
VMOVUPD __m128d _mm_mask_loadu_pd(__m128d s, __mmask8 k, void * m);
VMOVUPD __m128d _mm_maskz_loadu_pd( __mmask8 k, void * m);
VMOVUPD void _mm_mask_storeu_pd( void * d, __mmask8 k, __m128d a);
MOVUPD __m256d _mm256_loadu_pd (double * p);
MOVUPD void _mm256_storeu_pd( double *p, __m256d a);
MOVUPD __m128d _mm_loadu_pd (double * p);
MOVUPD void _mm_storeu_pd( double *p, __m128d a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

Note treatment of #AC varies; additionally:

#UD	If VEX.vvvv != 1111B.

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

MOVUPS—Move Unaligned Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 10 /r MOVUPS xmm1, xmm2/m128	A	V/V	SSE	Move unaligned packed single precision floating-point from xmm2/mem to xmm1.
NP 0F 11 /r MOVUPS xmm2/m128, xmm1	B	V/V	SSE	Move unaligned packed single precision floating-point from xmm1 to xmm2/mem.
VEX.128.0F.WIG 10 /r VMOVUPS xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed single precision floating-point from xmm2/mem to xmm1.
VEX.128.0F.WIG 11 /r VMOVUPS xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed single precision floating-point from xmm1 to xmm2/mem.
VEX.256.0F.WIG 10 /r VMOVUPS ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed single precision floating-point from ymm2/mem to ymm1.
VEX.256.0F.WIG 11 /r VMOVUPS ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed single precision floating-point from ymm1 to ymm2/mem.
EVEX.128.0F.W0 10 /r VMOVUPS xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed single precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.0F.W0 10 /r VMOVUPS ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed single precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.0F.W0 10 /r VMOVUPS zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1[1]	Move unaligned packed single precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.0F.W0 11 /r VMOVUPS xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed single precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.0F.W0 11 /r VMOVUPS ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Move unaligned packed single precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.0F.W0 11 /r VMOVUPS zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1[1]	Move unaligned packed single precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

EVEX.512 encoded version:

Moves 512 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into memory. The destination operand is updated according to the writemask.

VEX.256 and EVEX.256 encoded versions:

Moves 256 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned without causing a general-protection exception (#GP) to be generated.

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVUPS (EVEX Encoded Versions, Register-Copy Form)

```
(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[i+31:i]
        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
```

VMOVUPS (EVEX Encoded Versions, Store-Form)

```
(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[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR;
```

VMOVUPS (EVEX Encoded Versions, Load-Form)

```
(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[i+31:i]
        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
```

VMOVUPS (VEX.256 Encoded Version, Load - and Register Copy)

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

VMOVUPS (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVUPS (VEX.128 Encoded Version)

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

MOVUPS (128-bit Load- and Register-Copy- Form Legacy SSE Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVUPS (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVUPS __m512 _mm512_loadu_ps(void * s);
VMOVUPS __m512 _mm512_mask_loadu_ps(__m512 a, __mmask16 k, void * s);
VMOVUPS __m512 _mm512_maskz_loadu_ps(__mmask16 k, void * s);
VMOVUPS void _mm512_storeu_ps(void * d, __m512 a);
VMOVUPS void _mm512_mask_storeu_ps(void * d, __mmask8 k, __m512 a);
VMOVUPS __m256 _mm256_mask_loadu_ps(__m256 a, __mmask8 k, void * s);
VMOVUPS __m256 _mm256_maskz_loadu_ps(__mmask8 k, void * s);
VMOVUPS void _mm256_mask_storeu_ps(void * d, __mmask8 k, __m256 a);
VMOVUPS __m128 _mm_mask_loadu_ps(__m128 a, __mmask8 k, void * s);
VMOVUPS __m128 _mm_maskz_loadu_ps(__mmask8 k, void * s);
VMOVUPS void _mm_mask_storeu_ps(void * d, __mmask8 k, __m128 a);
MOVUPS __m256 _mm256_loadu_ps (float * p);
MOVUPS void _mm256 _storeu_ps(float *p, __m256 a);
MOVUPS __m128 _mm_loadu_ps (float * p);
MOVUPS void _mm_storeu_ps(float *p, __m128 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

Note treatment of #AC varies.

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

Additionally:

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

MULPD—Multiply Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 59 /r MULPD xmm1, xmm2/m128	A	V/V	SSE2	Multiply packed double precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.128.66.0F.WIG 59 /r VMULPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Multiply packed double precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.256.66.0F.WIG 59 /r VMULPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Multiply packed double precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.128.66.0F.W1 59 /r VMULPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Multiply packed double precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1.
EVEX.256.66.0F.W1 59 /r VMULPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Multiply packed double precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1.
EVEX.512.66.0F.W1 59 /r VMULPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F OR AVX10.1[1]	Multiply packed double precision floating-point values in zmm3/m512/m64bcst with zmm2 and store result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Multiply packed double precision floating-point values from the first source operand with corresponding values in the second source operand, and stores the packed double precision floating-point results in the destination operand.

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

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

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the destination YMM 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 ZMM register destination are unmodified.

Operation

VMULPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
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] * SRC2[63:0]
                ELSE
                    DEST[i+63:i] := 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
```

VMULPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] * SRC2[63:0]
DEST[127:64] := SRC1[127:64] * SRC2[127:64]
DEST[191:128] := SRC1[191:128] * SRC2[191:128]
DEST[255:192] := SRC1[255:192] * SRC2[255:192]
DEST[MAXVL-1:256] := 0;
.
```

VMULPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] * SRC2[63:0]
DEST[127:64] := SRC1[127:64] * SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

MULPD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] * SRC[63:0]
DEST[127:64] := DEST[127:64] * SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULPD __m512d _mm512_mul_pd(__m512d a, __m512d b);
VMULPD __m512d _mm512_mask_mul_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMULPD __m512d _mm512_maskz_mul_pd(__mmask8 k, __m512d a, __m512d b);
VMULPD __m512d _mm512_mul_round_pd(__m512d a, __m512d b, int);
VMULPD __m512d _mm512_mask_mul_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMULPD __m512d _mm512_maskz_mul_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VMULPD __m256d _mm256_mul_pd (__m256d a, __m256d b);
MULPD __m128d _mm_mul_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

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

MULPS—Multiply Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 59 /r MULPS xmm1, xmm2/m128	A	V/V	SSE	Multiply packed single precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.128.0F.WIG 59 /r VMULPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Multiply packed single precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.256.0F.WIG 59 /r VMULPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Multiply packed single precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.128.0F.W0 59 /r VMULPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Multiply packed single precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1.
EVEX.256.0F.W0 59 /r VMULPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Multiply packed single precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1.
EVEX.512.0F.W0 59 /r VMULPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	C	V/V	AVX512F OR AVX10.1[1]	Multiply packed single precision floating-point values in zmm3/m512/m32bcst with zmm2 and store result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Multiply the packed single precision floating-point values from the first source operand with the corresponding values in the second source operand, and stores the packed double precision floating-point results in the destination operand.

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

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

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the destination YMM 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 ZMM register destination are unmodified.

Operation

VMULPS (EVEX Encoded Version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
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] * SRC2[31:0]
                ELSE
                    DEST[i+31:i] := 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
```

VMULPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] * SRC2[31:0]
DEST[63:32] := SRC1[63:32] * SRC2[63:32]
DEST[95:64] := SRC1[95:64] * SRC2[95:64]
DEST[127:96] := SRC1[127:96] * SRC2[127:96]
DEST[159:128] := SRC1[159:128] * SRC2[159:128]
DEST[191:160] := SRC1[191:160] * SRC2[191:160]
DEST[223:192] := SRC1[223:192] * SRC2[223:192]
DEST[255:224] := SRC1[255:224] * SRC2[255:224].
DEST[MAXVL-1:256] := 0;
```

VMULPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] * SRC2[31:0]
DEST[63:32] := SRC1[63:32] * SRC2[63:32]
DEST[95:64] := SRC1[95:64] * SRC2[95:64]
DEST[127:96] := SRC1[127:96] * SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

MULPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0] * SRC2[31:0]
DEST[63:32] := SRC1[63:32] * SRC2[63:32]
DEST[95:64] := SRC1[95:64] * SRC2[95:64]
DEST[127:96] := SRC1[127:96] * SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULPS __m512 _mm512_mul_ps(__m512 a, __m512 b);
VMULPS __m512 _mm512_mask_mul_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_maskz_mul_ps(__mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_mul_round_ps(__m512 a, __m512 b, int);
VMULPS __m512 _mm512_mask_mul_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMULPS __m512 _mm512_maskz_mul_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMULPS __m256 _mm256_mask_mul_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMULPS __m256 _mm256_maskz_mul_ps(__mmask8 k, __m256 a, __m256 b);
VMULPS __m128 _mm_mask_mul_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULPS __m128 _mm_maskz_mul_ps(__mmask8 k, __m128 a, __m128 b);
VMULPS __m256 _mm256_mul_ps (__m256 a, __m256 b);
MULPS __m128 _mm_mul_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

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

MULSD—Multiply Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 59 /r MULSD xmm1,xmm2/m64	A	V/V	SSE2	Multiply the low double precision floating-point value in xmm2/m64 by low double precision floating-point value in xmm1.
VEX.LIG.F2.0F.WIG 59 /r VMULSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Multiply the low double precision floating-point value in xmm3/m64 by low double precision floating-point value in xmm2.
EVEX.LLIG.F2.0F.W1 59 /r VMULSD xmm1 {k1}{z}, xmm2, xmm3/m64 {er}	C	V/V	AVX512F OR AVX10.1[1]	Multiply the low double precision floating-point value in xmm3/m64 by low double precision floating-point value in xmm2.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Multiplies the low double precision floating-point value in the second source operand by the low double precision floating-point value in the first source operand, and stores the double precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The quadword at bits 127:64 of the destination operand is copied from the same bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the write-mask.

Software should ensure VMULSD is encoded with VEX.L=0. Encoding VMULSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMULSD (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC1[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;
ENDFOR
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VMULSD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] * SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

MULSD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] * SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULSD __m128d _mm_mask_mul_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_maskz_mul_sd(__mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_mul_round_sd(__m128d a, __m128d b, int);
VMULSD __m128d _mm_mask_mul_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMULSD __m128d _mm_maskz_mul_round_sd(__mmask8 k, __m128d a, __m128d b, int);
MULSD __m128d _mm_mul_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

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

MULSS—Multiply Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 59 /r MULSS xmm1,xmm2/m32	A	V/V	SSE	Multiply the low single precision floating-point value in xmm2/m32 by the low single precision floating-point value in xmm1.
VEX.LIG.F3.0F.WIG 59 /r VMULSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Multiply the low single precision floating-point value in xmm3/m32 by the low single precision floating-point value in xmm2.
EVEX.LLIG.F3.0F.W0 59 /r VMULSS xmm1 {k1}{z}, xmm2, xmm3/m32 {er}	C	V/V	AVX512F OR AVX10.1[1]	Multiply the low single precision floating-point value in xmm3/m32 by the low single precision floating-point value in xmm2.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Multiplies the low single precision floating-point value from the second source operand by the low single precision floating-point value in the first source operand, and stores the single precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the write-mask.

Software should ensure VMULSS is encoded with VEX.L=0. Encoding VMULSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMULSS (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC1[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;
ENDFOR
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VMULSS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] * SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

MULSS (128-bit Legacy SSE Version)

```
DEST[31:0] := DEST[31:0] * SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMULSS __m128 _mm_mask_mul_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_maskz_mul_ss( __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_mul_round_ss( __m128 a, __m128 b, int);
VMULSS __m128 _mm_mask_mul_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMULSS __m128 _mm_maskz_mul_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MULSS __m128 _mm_mul_ss(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

Underflow, Overflow, Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

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

ORPD—Bitwise Logical OR of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 56/r ORPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical OR of packed double precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F 56 /r VORPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical OR of packed double precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F 56 /r VORPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical OR of packed double precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 56 /r VORPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Return the bitwise logical OR of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 56 /r VORPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Return the bitwise logical OR of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 56 /r VORPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ OR AVX10.1[1]	Return the bitwise logical OR of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a bitwise logical OR 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 encoded versions: 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-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: 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. 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

VORPD (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 OR SRC2[63:0]
                ELSE
                    DEST[i+63:i] := SRC1[i+63:i] BITWISE OR 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
```

VORPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[191:128] := SRC1[191:128] BITWISE OR SRC2[191:128]
DEST[255:192] := SRC1[255:192] BITWISE OR SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VORPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

ORPD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] BITWISE OR SRC[63:0]
DEST[127:64] := DEST[127:64] BITWISE OR SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VORPD __m512d _mm512_or_pd ( __m512d a, __m512d  b);
VORPD __m512d _mm512_mask_or_pd ( __m512d s, __mmask8 k, __m512d a, __m512d b);
VORPD __m512d _mm512_maskz_or_pd (__mmask8 k, __m512d a, __m512d b);
VORPD __m256d _mm256_mask_or_pd (__m256d s, ___mmask8 k, __m256d a, __m256d b);
VORPD __m256d _mm256_maskz_or_pd (__mmask8 k, __m256d a, __m256d b);
VORPD __m128d _mm_mask_or_pd ( __m128d s, __mmask8 k, __m128d a, __m128d b);
VORPD __m128d _mm_maskz_or_pd (__mmask8 k, __m128d a, __m128d b);
VORPD __m256d _mm256_or_pd (__m256d a, __m256d b);
ORPD __m128d _mm_or_pd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

ORPS—Bitwise Logical OR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 56 /r ORPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical OR of packed single precision floating-point values in xmm1 and xmm2/mem.
VEX.128.0F 56 /r VORPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical OR of packed single precision floating-point values in xmm2 and xmm3/mem.
VEX.256.0F 56 /r VORPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical OR of packed single precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.0F.W0 56 /r VORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Return the bitwise logical OR of packed single precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.0F.W0 56 /r VORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Return the bitwise logical OR of packed single precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.0F.W0 56 /r VORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ OR AVX10.1[1]	Return the bitwise logical OR of packed single precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a bitwise logical OR 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 encoded versions: 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-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: 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. 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

VORPS (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 OR SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC1[i+31:i] BITWISE OR 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
```

VORPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[159:128] := SRC1[159:128] BITWISE OR SRC2[159:128]
DEST[191:160] := SRC1[191:160] BITWISE OR SRC2[191:160]
DEST[223:192] := SRC1[223:192] BITWISE OR SRC2[223:192]
DEST[255:224] := SRC1[255:224] BITWISE OR SRC2[255:224].
DEST[MAXVL-1:256] := 0
```

VORPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

ORPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VORPS __m512 _mm512_or_ps (__m512 a, __m512 b);
VORPS __m512 _mm512_mask_or_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VORPS __m512 _mm512_maskz_or_ps (__mmask16 k, __m512 a, __m512 b);
VORPS __m256 _mm256_mask_or_ps (__m256 s, ___mmask8 k, __m256 a, __m256 b);
VORPS __m256 _mm256_maskz_or_ps (__mmask8 k, __m256 a, __m256 b);
VORPS __m128 _mm_mask_or_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VORPS __m128 _mm_maskz_or_ps (__mmask8 k, __m128 a, __m128 b);
VORPS __m256 _mm256_or_ps (__m256 a, __m256 b);
ORPS __m128 _mm_or_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PABSB/PABSW/PABSD/PABSQ—Packed Absolute Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 1C /r[1] PABSB mm1, mm2/m64	A	V/V	SSSE3	Compute the absolute value of bytes in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1C /r PABSB xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
NP 0F 38 1D /r[1] PABSW mm1, mm2/m64	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1D /r PABSW xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
NP 0F 38 1E /r[1] PABSD mm1, mm2/m64	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1E /r PABSD xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1C /r VPABSB xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1D /r VPABSW xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of 16- bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1E /r VPABSD xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of 32- bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.256.66.0F38.WIG 1C /r VPABSB ymm1, ymm2/m256	A	V/V	AVX2	Compute the absolute value of bytes in ymm2/m256 and store UNSIGNED result in ymm1.
VEX.256.66.0F38.WIG 1D /r VPABSW ymm1, ymm2/m256	A	V/V	AVX2	Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.
VEX.256.66.0F38.WIG 1E /r VPABSD ymm1, ymm2/m256	A	V/V	AVX2	Compute the absolute value of 32-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.
EVEX.128.66.0F38.WIG 1C /r VPABSB xmm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.WIG 1C /r VPABSB ymm1 {k1}{z}, ymm2/m256	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compute the absolute value of bytes in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.WIG 1C /r VPABSB zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512BW OR AVX10.1[2]	Compute the absolute value of bytes in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.WIG 1D /r VPABSW xmm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.WIG 1D /r VPABSW ymm1 {k1}{z}, ymm2/m256	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.WIG 1D /r VPABSW zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512BW OR AVX10.1[2]	Compute the absolute value of 16-bit integers in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EVEX.128.66.0F38.W0 1E /r VPABSD xmm1 {k1}{z}, xmm2/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Compute the absolute value of 32-bit integers in xmm2/m128/m32bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 1E /r VPABSD ymm1 {k1}{z}, ymm2/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Compute the absolute value of 32-bit integers in ymm2/m256/m32bcst and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 1E /r VPABSD zmm1 {k1}{z}, zmm2/m512/m32bcst	C	V/V	AVX512F OR AVX10.1[2]	Compute the absolute value of 32-bit integers in zmm2/m512/m32bcst and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 1F /r VPABSQ xmm1 {k1}{z}, xmm2/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Compute the absolute value of 64-bit integers in xmm2/m128/m64bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 1F /r VPABSQ ymm1 {k1}{z}, ymm2/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Compute the absolute value of 64-bit integers in ymm2/m256/m64bcst and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 1F /r VPABSQ zmm1 {k1}{z}, zmm2/m512/m64bcst	C	V/V	AVX512F OR AVX10.1[2]	Compute the absolute value of 64-bit integers in zmm2/m512/m64bcst and store UNSIGNED result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). PABSB operates on signed bytes, PABSW operates on signed 16-bit words, and PABSD operates on signed 32-bit integers.

EVEX encoded VPABSD/Q: 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 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPABSB/W: 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 updated according to the writemask.

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

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

128-bit Legacy SSE version: The source operand can be an XMM register or an 128-bit memory location. The destination is an XMM register. The upper bits (VL_MAX-1:128) of the corresponding register destination are unmodified.

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

Operation

PABSB With 128-bit Operands:

```
    Unsigned DEST[7:0] := ABS(SRC[7: 0])
    Repeat operation for 2nd through 15th bytes
    Unsigned DEST[127:120] := ABS(SRC[127:120])
```

VPABSB With 128-bit Operands:

```
    Unsigned DEST[7:0] := ABS(SRC[7: 0])
    Repeat operation for 2nd through 15th bytes
    Unsigned DEST[127:120] := ABS(SRC[127:120])
```

VPABSB With 256-bit Operands:

```
    Unsigned DEST[7:0] := ABS(SRC[7: 0])
    Repeat operation for 2nd through 31st bytes
    Unsigned DEST[255:248] := ABS(SRC[255:248])
```

VPABSB (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
            Unsigned DEST[i+7:i] := ABS(SRC[i+7:i])
        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] := 0
```

PABSW With 128-bit Operands:

```
    Unsigned DEST[15:0] := ABS(SRC[15:0])
    Repeat operation for 2nd through 7th 16-bit words
    Unsigned DEST[127:112] := ABS(SRC[127:112])
```

VPABSW With 128-bit Operands:

```
    Unsigned DEST[15:0] := ABS(SRC[15:0])
    Repeat operation for 2nd through 7th 16-bit words
    Unsigned DEST[127:112] := ABS(SRC[127:112])
```

VPABSW With 256-bit Operands:

```
    Unsigned DEST[15:0] := ABS(SRC[15:0])
    Repeat operation for 2nd through 15th 16-bit words
    Unsigned DEST[255:240] := ABS(SRC[255:240])
```

VPABSW (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
            Unsigned DEST[i+15:i] := ABS(SRC[i+15:i])
        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] := 0
```

PABSD With 128-bit Operands:

```
    Unsigned DEST[31:0] := ABS(SRC[31:0])
    Repeat operation for 2nd through 3rd 32-bit double words
    Unsigned DEST[127:96] := ABS(SRC[127:96])
```

VPABSD With 128-bit Operands:

```
    Unsigned DEST[31:0] := ABS(SRC[31:0])
    Repeat operation for 2nd through 3rd 32-bit double words
    Unsigned DEST[127:96] := ABS(SRC[127:96])
```

VPABSD With 256-bit Operands:

```
    Unsigned DEST[31:0] := ABS(SRC[31:0])
    Repeat operation for 2nd through 7th 32-bit double words
    Unsigned DEST[255:224] := ABS(SRC[255:224])
```

VPABSD (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
                    Unsigned DEST[i+31:i] := ABS(SRC[31:0])
                ELSE
                    Unsigned DEST[i+31:i] := ABS(SRC[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
```

VPABSQ (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
                    Unsigned DEST[i+63:i] := ABS(SRC[63:0])
                ELSE
                    Unsigned DEST[i+63:i] := ABS(SRC[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 Equivalents

```
VPABSB__m512i _mm512_abs_epi8 ( __m512i a)
VPABSW__m512i _mm512_abs_epi16 ( __m512i a)
VPABSB__m512i _mm512_mask_abs_epi8 ( __m512i s, __mmask64 m, __m512i a)
VPABSW__m512i _mm512_mask_abs_epi16 ( __m512i s, __mmask32 m, __m512i a)
VPABSB__m512i _mm512_maskz_abs_epi8 (__mmask64 m, __m512i a)
VPABSW__m512i _mm512_maskz_abs_epi16 (__mmask32 m, __m512i a)
VPABSB__m256i _mm256_mask_abs_epi8 (__m256i s, __mmask32 m, __m256i a)
VPABSW__m256i _mm256_mask_abs_epi16 (__m256i s, __mmask16 m, __m256i a)
VPABSB__m256i _mm256_maskz_abs_epi8 (__mmask32 m, __m256i a)
VPABSW__m256i _mm256_maskz_abs_epi16 (__mmask16 m, __m256i a)
VPABSB__m128i _mm_mask_abs_epi8 (__m128i s, __mmask16 m, __m128i a)
VPABSW__m128i _mm_mask_abs_epi16 (__m128i s, __mmask8 m, __m128i a)
VPABSB__m128i _mm_maskz_abs_epi8 (__mmask16 m, __m128i a)
VPABSW__m128i _mm_maskz_abs_epi16 (__mmask8 m, __m128i a)
VPABSD __m256i _mm256_mask_abs_epi32(__m256i s, __mmask8 k, __m256i a);
VPABSD __m256i _mm256_maskz_abs_epi32( __mmask8 k, __m256i a);
VPABSD __m128i _mm_mask_abs_epi32(__m128i s, __mmask8 k, __m128i a);
VPABSD __m128i _mm_maskz_abs_epi32( __mmask8 k, __m128i a);
VPABSD  __m512i _mm512_abs_epi32( __m512i a);
VPABSD  __m512i _mm512_mask_abs_epi32(__m512i s, __mmask16 k, __m512i a);
VPABSD  __m512i _mm512_maskz_abs_epi32( __mmask16 k, __m512i a);
VPABSQ __m512i _mm512_abs_epi64( __m512i a);
VPABSQ __m512i _mm512_mask_abs_epi64(__m512i s, __mmask8 k, __m512i a);
VPABSQ __m512i _mm512_maskz_abs_epi64( __mmask8 k, __m512i a);
VPABSQ __m256i _mm256_mask_abs_epi64(__m256i s, __mmask8 k, __m256i a);
VPABSQ __m256i _mm256_maskz_abs_epi64( __mmask8 k, __m256i a);
VPABSQ __m128i _mm_mask_abs_epi64(__m128i s, __mmask8 k, __m128i a);
VPABSQ __m128i _mm_maskz_abs_epi64( __mmask8 k, __m128i a);
PABSB __m128i _mm_abs_epi8 (__m128i a)
VPABSB __m128i _mm_abs_epi8 (__m128i a)
```

VPABSB __m256i _mm256_abs_epi8 (__m256i a)
PABSW __m128i _mm_abs_epi16 (__m128i a)
VPABSW __m128i _mm_abs_epi16 (__m128i a)
VPABSW __m256i _mm256_abs_epi16 (__m256i a)
PABSD __m128i _mm_abs_epi32 (__m128i a)
VPABSD __m128i _mm_abs_epi32 (__m128i a)
VPABSD __m256i _mm256_abs_epi32 (__m256i a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPABSD/Q, see Table 2-49, “Type E4 Class Exception Conditions.”

EVEX-encoded VPABSB/W, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

PACKSSWB/PACKSSDW—Pack With Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 63 /r[1] PACKSSWB mm1, mm2/m64	A	V/V	MMX	Converts 4 packed signed word integers from mm1 and from mm2/m64 into 8 packed signed byte integers in mm1 using signed saturation.
66 0F 63 /r PACKSSWB xmm1, xmm2/m128	A	V/V	SSE2	Converts 8 packed signed word integers from xmm1 and from xmm2/m128 into 16 packed signed byte integers in xmm1 using signed saturation.
NP 0F 6B /r[1] PACKSSDW mm1, mm2/m64	A	V/V	MMX	Converts 2 packed signed doubleword integers from mm1 and from mm2/m64 into 4 packed signed word integers in mm1 using signed saturation.
66 0F 6B /r PACKSSDW xmm1, xmm2/m128	A	V/V	SSE2	Converts 4 packed signed doubleword integers from xmm1 and from xmm2/m128 into 8 packed signed word integers in xmm1 using signed saturation.
VEX.128.66.0F.WIG 63 /r VPACKSSWB xmm1,xmm2, xmm3/m128	B	V/V	AVX	Converts 8 packed signed word integers from xmm2 and from xmm3/m128 into 16 packed signed byte integers in xmm1 using signed saturation.
VEX.128.66.0F.WIG 6B /r VPACKSSDW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Converts 4 packed signed doubleword integers from xmm2 and from xmm3/m128 into 8 packed signed word integers in xmm1 using signed saturation.
VEX.256.66.0F.WIG 63 /r VPACKSSWB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Converts 16 packed signed word integers from ymm2 and from ymm3/m256 into 32 packed signed byte integers in ymm1 using signed saturation.
VEX.256.66.0F.WIG 6B /r VPACKSSDW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Converts 8 packed signed doubleword integers from ymm2 and from ymm3/m256 into 16 packed signed word integers in ymm1using signed saturation.
EVEX.128.66.0F.WIG 63 /r VPACKSSWB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Converts packed signed word integers from xmm2 and from xmm3/m128 into packed signed byte integers in xmm1 using signed saturation under writemask k1.
EVEX.256.66.0F.WIG 63 /r VPACKSSWB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Converts packed signed word integers from ymm2 and from ymm3/m256 into packed signed byte integers in ymm1 using signed saturation under writemask k1.
EVEX.512.66.0F.WIG 63 /r VPACKSSWB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Converts packed signed word integers from zmm2 and from zmm3/m512 into packed signed byte integers in zmm1 using signed saturation under writemask k1.
EVEX.128.66.0F.W0 6B /r VPACKSSDW xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Converts packed signed doubleword integers from xmm2 and from xmm3/m128/m32bcst into packed signed word integers in xmm1 using signed saturation under writemask k1.

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EVEX.256.66.0F.W0 6B /r VPACKSSDW ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Converts packed signed doubleword integers from ymm2 and from ymm3/m256/m32bcst into packed signed word integers in ymm1 using signed saturation under writemask k1.
EVEX.512.66.0F.W0 6B /r VPACKSSDW zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512BW OR AVX10.1[2]	Converts packed signed doubleword integers from zmm2 and from zmm3/m512/m32bcst into packed signed word integers in zmm1 using signed saturation under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 1-9 for an example of the packing operation.

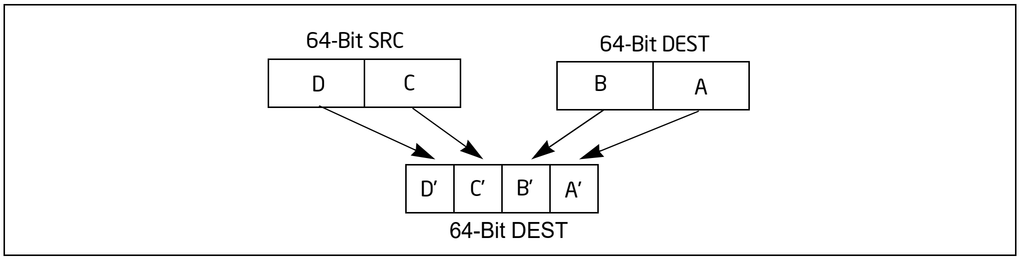


Figure 1-9. Operation of the PACKSSDW Instruction Using 64-Bit Operands

PACKSSWB converts packed signed word integers in the first and second source operands into packed signed byte integers using signed saturation to handle overflow conditions beyond the range of signed byte integers. If the signed word value is beyond the range of a signed byte value (i.e., greater than 7FH or less than 80H), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination. PACKSSDW converts packed signed doubleword integers in the first and second source operands into packed signed word integers using signed saturation to handle overflow conditions beyond 7FFFH and 8000H.

EVEX encoded PACKSSWB: 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 a ZMM/YMM/XMM register, updated conditional under the writemask k1.

EVEX encoded PACKSSDW: 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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask k1.

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. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: 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. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand 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 ZMM destination register destination are unmodified.

Operation

PACKSSWB Instruction (128-bit Legacy SSE Version)

```
    DEST[7:0] := SaturateSignedWordToSignedByte (DEST[15:0]);
    DEST[15:8] := SaturateSignedWordToSignedByte (DEST[31:16]);
    DEST[23:16] := SaturateSignedWordToSignedByte (DEST[47:32]);
    DEST[31:24] := SaturateSignedWordToSignedByte (DEST[63:48]);
    DEST[39:32] := SaturateSignedWordToSignedByte (DEST[79:64]);
    DEST[47:40] := SaturateSignedWordToSignedByte (DEST[95:80]);
    DEST[55:48] := SaturateSignedWordToSignedByte (DEST[111:96]);
    DEST[63:56] := SaturateSignedWordToSignedByte (DEST[127:112]);
    DEST[71:64] := SaturateSignedWordToSignedByte (SRC[15:0]);
    DEST[79:72] := SaturateSignedWordToSignedByte (SRC[31:16]);
    DEST[87:80] := SaturateSignedWordToSignedByte (SRC[47:32]);
    DEST[95:88] := SaturateSignedWordToSignedByte (SRC[63:48]);
    DEST[103:96] := SaturateSignedWordToSignedByte (SRC[79:64]);
    DEST[111:104] := SaturateSignedWordToSignedByte (SRC[95:80]);
    DEST[119:112] := SaturateSignedWordToSignedByte (SRC[111:96]);
    DEST[127:120] := SaturateSignedWordToSignedByte (SRC[127:112]);
    DEST[MAXVL-1:128] (Unmodified)
```

PACKSSDW Instruction (128-bit Legacy SSE Version)

```
    DEST[15:0] := SaturateSignedDwordToSignedWord (DEST[31:0]);
    DEST[31:16] := SaturateSignedDwordToSignedWord (DEST[63:32]);
    DEST[47:32] := SaturateSignedDwordToSignedWord (DEST[95:64]);
    DEST[63:48] := SaturateSignedDwordToSignedWord (DEST[127:96]);
    DEST[79:64] := SaturateSignedDwordToSignedWord (SRC[31:0]);
    DEST[95:80] := SaturateSignedDwordToSignedWord (SRC[63:32]);
    DEST[111:96] := SaturateSignedDwordToSignedWord (SRC[95:64]);
    DEST[127:112] := SaturateSignedDwordToSignedWord (SRC[127:96]);
    DEST[MAXVL-1:128] (Unmodified)
```

VPACKSSWB Instruction (VEX.128 Encoded Version)

```
    DEST[7:0] := SaturateSignedWordToSignedByte (SRC1[15:0]);
    DEST[15:8] := SaturateSignedWordToSignedByte (SRC1[31:16]);
    DEST[23:16] := SaturateSignedWordToSignedByte (SRC1[47:32]);
    DEST[31:24] := SaturateSignedWordToSignedByte (SRC1[63:48]);
    DEST[39:32] := SaturateSignedWordToSignedByte (SRC1[79:64]);
    DEST[47:40] := SaturateSignedWordToSignedByte (SRC1[95:80]);
    DEST[55:48] := SaturateSignedWordToSignedByte (SRC1[111:96]);
    DEST[63:56] := SaturateSignedWordToSignedByte (SRC1[127:112]);
    DEST[71:64] := SaturateSignedWordToSignedByte (SRC2[15:0]);
    DEST[79:72] := SaturateSignedWordToSignedByte (SRC2[31:16]);
    DEST[87:80] := SaturateSignedWordToSignedByte (SRC2[47:32]);
    DEST[95:88] := SaturateSignedWordToSignedByte (SRC2[63:48]);
    DEST[103:96] := SaturateSignedWordToSignedByte (SRC2[79:64]);
    DEST[111:104] := SaturateSignedWordToSignedByte (SRC2[95:80]);
    DEST[119:112] := SaturateSignedWordToSignedByte (SRC2[111:96]);
    DEST[127:120] := SaturateSignedWordToSignedByte (SRC2[127:112]);
    DEST[MAXVL-1:128] := 0;
```

VPACKSSDW Instruction (VEX.128 Encoded Version)

```
    DEST[15:0] := SaturateSignedDwordToSignedWord (SRC1[31:0]);
    DEST[31:16] := SaturateSignedDwordToSignedWord (SRC1[63:32]);
    DEST[47:32] := SaturateSignedDwordToSignedWord (SRC1[95:64]);
    DEST[63:48] := SaturateSignedDwordToSignedWord (SRC1[127:96]);
    DEST[79:64] := SaturateSignedDwordToSignedWord (SRC2[31:0]);
    DEST[95:80] := SaturateSignedDwordToSignedWord (SRC2[63:32]);
    DEST[111:96] := SaturateSignedDwordToSignedWord (SRC2[95:64]);
    DEST[127:112] := SaturateSignedDwordToSignedWord (SRC2[127:96]);
    DEST[MAXVL-1:128] := 0;
```

VPACKSSWB Instruction (VEX.256 Encoded Version)

```
    DEST[7:0] := SaturateSignedWordToSignedByte (SRC1[15:0]);
    DEST[15:8] := SaturateSignedWordToSignedByte (SRC1[31:16]);
    DEST[23:16] := SaturateSignedWordToSignedByte (SRC1[47:32]);
    DEST[31:24] := SaturateSignedWordToSignedByte (SRC1[63:48]);
    DEST[39:32] := SaturateSignedWordToSignedByte (SRC1[79:64]);
    DEST[47:40] := SaturateSignedWordToSignedByte (SRC1[95:80]);
    DEST[55:48] := SaturateSignedWordToSignedByte (SRC1[111:96]);
    DEST[63:56] := SaturateSignedWordToSignedByte (SRC1[127:112]);
    DEST[71:64] := SaturateSignedWordToSignedByte (SRC2[15:0]);
    DEST[79:72] := SaturateSignedWordToSignedByte (SRC2[31:16]);
    DEST[87:80] := SaturateSignedWordToSignedByte (SRC2[47:32]);
    DEST[95:88] := SaturateSignedWordToSignedByte (SRC2[63:48]);
    DEST[103:96] := SaturateSignedWordToSignedByte (SRC2[79:64]);
    DEST[111:104] := SaturateSignedWordToSignedByte (SRC2[95:80]);
    DEST[119:112] := SaturateSignedWordToSignedByte (SRC2[111:96]);
    DEST[127:120] := SaturateSignedWordToSignedByte (SRC2[127:112]);
    DEST[135:128] := SaturateSignedWordToSignedByte (SRC1[143:128]);
    DEST[143:136] := SaturateSignedWordToSignedByte (SRC1[159:144]);
    DEST[151:144] := SaturateSignedWordToSignedByte (SRC1[175:160]);
    DEST[159:152] := SaturateSignedWordToSignedByte (SRC1[191:176]);
    DEST[167:160] := SaturateSignedWordToSignedByte (SRC1[207:192]);
    DEST[175:168] := SaturateSignedWordToSignedByte (SRC1[223:208]);
    DEST[183:176] := SaturateSignedWordToSignedByte (SRC1[239:224]);
```

```
    DEST[191:184] := SaturateSignedWordToSignedByte (SRC1[255:240]);
    DEST[199:192] := SaturateSignedWordToSignedByte (SRC2[143:128]);
    DEST[207:200] := SaturateSignedWordToSignedByte (SRC2[159:144]);
    DEST[215:208] := SaturateSignedWordToSignedByte (SRC2[175:160]);
    DEST[223:216] := SaturateSignedWordToSignedByte (SRC2[191:176]);
    DEST[231:224] := SaturateSignedWordToSignedByte (SRC2[207:192]);
    DEST[239:232] := SaturateSignedWordToSignedByte (SRC2[223:208]);
    DEST[247:240] := SaturateSignedWordToSignedByte (SRC2[239:224]);
    DEST[255:248] := SaturateSignedWordToSignedByte (SRC2[255:240]);
    DEST[MAXVL-1:256] := 0;
```

VPACKSSDW Instruction (VEX.256 Encoded Version)

```
    DEST[15:0] := SaturateSignedDwordToSignedWord (SRC1[31:0]);
    DEST[31:16] := SaturateSignedDwordToSignedWord (SRC1[63:32]);
    DEST[47:32] := SaturateSignedDwordToSignedWord (SRC1[95:64]);
    DEST[63:48] := SaturateSignedDwordToSignedWord (SRC1[127:96]);
    DEST[79:64] := SaturateSignedDwordToSignedWord (SRC2[31:0]);
    DEST[95:80] := SaturateSignedDwordToSignedWord (SRC2[63:32]);
    DEST[111:96] := SaturateSignedDwordToSignedWord (SRC2[95:64]);
    DEST[127:112] := SaturateSignedDwordToSignedWord (SRC2[127:96]);
    DEST[143:128] := SaturateSignedDwordToSignedWord (SRC1[159:128]);
    DEST[159:144] := SaturateSignedDwordToSignedWord (SRC1[191:160]);
    DEST[175:160] := SaturateSignedDwordToSignedWord (SRC1[223:192]);
    DEST[191:176] := SaturateSignedDwordToSignedWord (SRC1[255:224]);
    DEST[207:192] := SaturateSignedDwordToSignedWord (SRC2[159:128]);
    DEST[223:208] := SaturateSignedDwordToSignedWord (SRC2[191:160]);
    DEST[239:224] := SaturateSignedDwordToSignedWord (SRC2[223:192]);
    DEST[255:240] := SaturateSignedDwordToSignedWord (SRC2[255:224]);
    DEST[MAXVL-1:256] := 0;
```

VPACKSSWB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] := SaturateSignedWordToSignedByte (SRC1[15:0]);
TMP_DEST[15:8] := SaturateSignedWordToSignedByte (SRC1[31:16]);
TMP_DEST[23:16] := SaturateSignedWordToSignedByte (SRC1[47:32]);
TMP_DEST[31:24] := SaturateSignedWordToSignedByte (SRC1[63:48]);
TMP_DEST[39:32] := SaturateSignedWordToSignedByte (SRC1[79:64]);
TMP_DEST[47:40] := SaturateSignedWordToSignedByte (SRC1[95:80]);
TMP_DEST[55:48] := SaturateSignedWordToSignedByte (SRC1[111:96]);
TMP_DEST[63:56] := SaturateSignedWordToSignedByte (SRC1[127:112]);
TMP_DEST[71:64] := SaturateSignedWordToSignedByte (SRC2[15:0]);
TMP_DEST[79:72] := SaturateSignedWordToSignedByte (SRC2[31:16]);
TMP_DEST[87:80] := SaturateSignedWordToSignedByte (SRC2[47:32]);
TMP_DEST[95:88] := SaturateSignedWordToSignedByte (SRC2[63:48]);
TMP_DEST[103:96] := SaturateSignedWordToSignedByte (SRC2[79:64]);
TMP_DEST[111:104] := SaturateSignedWordToSignedByte (SRC2[95:80]);
TMP_DEST[119:112] := SaturateSignedWordToSignedByte (SRC2[111:96]);
TMP_DEST[127:120] := SaturateSignedWordToSignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128] := SaturateSignedWordToSignedByte (SRC1[143:128]);
    TMP_DEST[143:136] := SaturateSignedWordToSignedByte (SRC1[159:144]);
    TMP_DEST[151:144] := SaturateSignedWordToSignedByte (SRC1[175:160]);
    TMP_DEST[159:152] := SaturateSignedWordToSignedByte (SRC1[191:176]);
    TMP_DEST[167:160] := SaturateSignedWordToSignedByte (SRC1[207:192]);
```

```
    TMP_DEST[175:168] := SaturateSignedWordToSignedByte (SRC1[223:208]);
    TMP_DEST[183:176] := SaturateSignedWordToSignedByte (SRC1[239:224]);
    TMP_DEST[191:184] := SaturateSignedWordToSignedByte (SRC1[255:240]);
    TMP_DEST[199:192] := SaturateSignedWordToSignedByte (SRC2[143:128]);
    TMP_DEST[207:200] := SaturateSignedWordToSignedByte (SRC2[159:144]);
    TMP_DEST[215:208] := SaturateSignedWordToSignedByte (SRC2[175:160]);
    TMP_DEST[223:216] := SaturateSignedWordToSignedByte (SRC2[191:176]);
    TMP_DEST[231:224] := SaturateSignedWordToSignedByte (SRC2[207:192]);
    TMP_DEST[239:232] := SaturateSignedWordToSignedByte (SRC2[223:208]);
    TMP_DEST[247:240] := SaturateSignedWordToSignedByte (SRC2[239:224]);
    TMP_DEST[255:248] := SaturateSignedWordToSignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] := SaturateSignedWordToSignedByte (SRC1[271:256]);
    TMP_DEST[271:264] := SaturateSignedWordToSignedByte (SRC1[287:272]);
    TMP_DEST[279:272] := SaturateSignedWordToSignedByte (SRC1[303:288]);
    TMP_DEST[287:280] := SaturateSignedWordToSignedByte (SRC1[319:304]);
    TMP_DEST[295:288] := SaturateSignedWordToSignedByte (SRC1[335:320]);
    TMP_DEST[303:296] := SaturateSignedWordToSignedByte (SRC1[351:336]);
    TMP_DEST[311:304] := SaturateSignedWordToSignedByte (SRC1[367:352]);
    TMP_DEST[319:312] := SaturateSignedWordToSignedByte (SRC1[383:368]);

    TMP_DEST[327:320] := SaturateSignedWordToSignedByte (SRC2[271:256]);
    TMP_DEST[335:328] := SaturateSignedWordToSignedByte (SRC2[287:272]);
    TMP_DEST[343:336] := SaturateSignedWordToSignedByte (SRC2[303:288]);
    TMP_DEST[351:344] := SaturateSignedWordToSignedByte (SRC2[319:304]);
    TMP_DEST[359:352] := SaturateSignedWordToSignedByte (SRC2[335:320]);
    TMP_DEST[367:360] := SaturateSignedWordToSignedByte (SRC2[351:336]);
    TMP_DEST[375:368] := SaturateSignedWordToSignedByte (SRC2[367:352]);
    TMP_DEST[383:376] := SaturateSignedWordToSignedByte (SRC2[383:368]);

    TMP_DEST[391:384] := SaturateSignedWordToSignedByte (SRC1[399:384]);
    TMP_DEST[399:392] := SaturateSignedWordToSignedByte (SRC1[415:400]);
    TMP_DEST[407:400] := SaturateSignedWordToSignedByte (SRC1[431:416]);
    TMP_DEST[415:408] := SaturateSignedWordToSignedByte (SRC1[447:432]);
    TMP_DEST[423:416] := SaturateSignedWordToSignedByte (SRC1[463:448]);
    TMP_DEST[431:424] := SaturateSignedWordToSignedByte (SRC1[479:464]);
    TMP_DEST[439:432] := SaturateSignedWordToSignedByte (SRC1[495:480]);
    TMP_DEST[447:440] := SaturateSignedWordToSignedByte (SRC1[511:496]);

    TMP_DEST[455:448] := SaturateSignedWordToSignedByte (SRC2[399:384]);
    TMP_DEST[463:456] := SaturateSignedWordToSignedByte (SRC2[415:400]);
    TMP_DEST[471:464] := SaturateSignedWordToSignedByte (SRC2[431:416]);
    TMP_DEST[479:472] := SaturateSignedWordToSignedByte (SRC2[447:432]);
    TMP_DEST[487:480] := SaturateSignedWordToSignedByte (SRC2[463:448]);
    TMP_DEST[495:488] := SaturateSignedWordToSignedByte (SRC2[479:464]);
    TMP_DEST[503:496] := SaturateSignedWordToSignedByte (SRC2[495:480]);
    TMP_DEST[511:504] := SaturateSignedWordToSignedByte (SRC2[511:496]);
FI;
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] := TMP_DEST[i+7:i]
```

```
            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] := 0
```

VPACKSSDW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO ((KL/2) - 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[15:0] := SaturateSignedDwordToSignedWord (SRC1[31:0]);
TMP_DEST[31:16] := SaturateSignedDwordToSignedWord (SRC1[63:32]);
TMP_DEST[47:32] := SaturateSignedDwordToSignedWord (SRC1[95:64]);
TMP_DEST[63:48] := SaturateSignedDwordToSignedWord (SRC1[127:96]);
TMP_DEST[79:64] := SaturateSignedDwordToSignedWord (TMP_SRC2[31:0]);
TMP_DEST[95:80] := SaturateSignedDwordToSignedWord (TMP_SRC2[63:32]);
TMP_DEST[111:96] := SaturateSignedDwordToSignedWord (TMP_SRC2[95:64]);
TMP_DEST[127:112] := SaturateSignedDwordToSignedWord (TMP_SRC2[127:96]);
IF VL >= 256
      TMP_DEST[143:128] := SaturateSignedDwordToSignedWord (SRC1[159:128]);
      TMP_DEST[159:144] := SaturateSignedDwordToSignedWord (SRC1[191:160]);
      TMP_DEST[175:160] := SaturateSignedDwordToSignedWord (SRC1[223:192]);
      TMP_DEST[191:176] := SaturateSignedDwordToSignedWord (SRC1[255:224]);
      TMP_DEST[207:192] := SaturateSignedDwordToSignedWord (TMP_SRC2[159:128]);
      TMP_DEST[223:208] := SaturateSignedDwordToSignedWord (TMP_SRC2[191:160]);
      TMP_DEST[239:224] := SaturateSignedDwordToSignedWord (TMP_SRC2[223:192]);
      TMP_DEST[255:240] := SaturateSignedDwordToSignedWord (TMP_SRC2[255:224]);
FI;
IF VL >= 512
      TMP_DEST[271:256] := SaturateSignedDwordToSignedWord (SRC1[287:256]);
      TMP_DEST[287:272] := SaturateSignedDwordToSignedWord (SRC1[319:288]);
      TMP_DEST[303:288] := SaturateSignedDwordToSignedWord (SRC1[351:320]);
      TMP_DEST[319:304] := SaturateSignedDwordToSignedWord (SRC1[383:352]);
      TMP_DEST[335:320] := SaturateSignedDwordToSignedWord (TMP_SRC2[287:256]);
      TMP_DEST[351:336] := SaturateSignedDwordToSignedWord (TMP_SRC2[319:288]);
      TMP_DEST[367:352] := SaturateSignedDwordToSignedWord (TMP_SRC2[351:320]);
      TMP_DEST[383:368] := SaturateSignedDwordToSignedWord (TMP_SRC2[383:352]);

      TMP_DEST[399:384] := SaturateSignedDwordToSignedWord (SRC1[415:384]);
      TMP_DEST[415:400] := SaturateSignedDwordToSignedWord (SRC1[447:416]);
      TMP_DEST[431:416] := SaturateSignedDwordToSignedWord (SRC1[479:448]);
```

```
    TMP_DEST[447:432] := SaturateSignedDwordToSignedWord (SRC1[511:480]);
    TMP_DEST[463:448] := SaturateSignedDwordToSignedWord (TMP_SRC2[415:384]);
    TMP_DEST[479:464] := SaturateSignedDwordToSignedWord (TMP_SRC2[447:416]);
    TMP_DEST[495:480] := SaturateSignedDwordToSignedWord (TMP_SRC2[479:448]);
    TMP_DEST[511:496] := SaturateSignedDwordToSignedWord (TMP_SRC2[511:480]);
FI;
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*           ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPACKSSDW__m512i _mm512_packs_epi32(__m512i m1, __m512i m2);
VPACKSSDW__m512i _mm512_mask_packs_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW__m512i _mm512_maskz_packs_epi32(__mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW__m256i _mm256_mask_packs_epi32(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW__m256i _mm256_maskz_packs_epi32(__mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW__m128i _mm_mask_packs_epi32(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSDW__m128i _mm_maskz_packs_epi32(__mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB__m512i _mm512_packs_epi16(__m512i m1, __m512i m2);
VPACKSSWB__m512i _mm512_mask_packs_epi16(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB__m512i _mm512_maskz_packs_epi16(__mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB__m256i _mm256_mask_packs_epi16(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB__m256i _mm256_maskz_packs_epi16(__mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB__m128i _mm_mask_packs_epi16(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB__m128i _mm_maskz_packs_epi16(__mmask8 k, __m128i m1, __m128i m2);
PACKSSWB __m128i _mm_packs_epi16(__m128i m1, __m128i m2)
PACKSSDW __m128i _mm_packs_epi32(__m128i m1, __m128i m2)
VPACKSSWB __m256i _mm256_packs_epi16(__m256i m1, __m256i m2)
VPACKSSDW __m256i _mm256_packs_epi32(__m256i m1, __m256i m2)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPACKSSDW, see Table 2-50, “Type E4NF Class Exception Conditions.”

EVEX-encoded VPACKSSWB, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

PACKUSDW—Pack With Unsigned Saturation

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2B /r PACKUSDW xmm1, xmm2/m128	A	V/V	SSE4_1	Convert 4 packed signed doubleword integers from xmm1 and 4 packed signed doubleword integers from xmm2/m128 into 8 packed unsigned word integers in xmm1 using unsigned saturation.
VEX.128.66.0F38 2B /r VPACKUSDW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Convert 4 packed signed doubleword integers from xmm2 and 4 packed signed doubleword integers from xmm3/m128 into 8 packed unsigned word integers in xmm1 using unsigned saturation.
VEX.256.66.0F38 2B /r VPACKUSDW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Convert 8 packed signed doubleword integers from ymm2 and 8 packed signed doubleword integers from ymm3/m256 into 16 packed unsigned word integers in ymm1 using unsigned saturation.
EVEX.128.66.0F38.W0 2B /r VPACKUSDW xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Convert packed signed doubleword integers from xmm2 and packed signed doubleword integers from xmm3/m128/m32bcst into packed unsigned word integers in xmm1 using unsigned saturation under writemask k1.
EVEX.256.66.0F38.W0 2B /r VPACKUSDW ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Convert packed signed doubleword integers from ymm2 and packed signed doubleword integers from ymm3/m256/m32bcst into packed unsigned word integers in ymm1 using unsigned saturation under writemask k1.
EVEX.512.66.0F38.W0 2B /r VPACKUSDW zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512BW OR AVX10.1[1]	Convert packed signed doubleword integers from zmm2 and packed signed doubleword integers from zmm3/m512/m32bcst into packed unsigned word integers in zmm1 using unsigned saturation under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Converts packed signed doubleword integers in the first and second source operands into packed unsigned word integers using unsigned saturation to handle overflow conditions. If the signed doubleword value is beyond the range of an unsigned word (that is, greater than FFFFH or less than 0000H), the saturated unsigned word integer value of FFFFH or 0000H, respectively, is stored in the destination.

EVEX encoded versions: 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 32-bit memory location. The destination operand is a ZMM register, updated conditionally under the writemask k1.

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. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: 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. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand 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 destination register destination are unmodified.

Operation

PACKUSDW (Legacy SSE Instruction)

```
TMP[15:0] := (DEST[31:0] < 0) ? 0 : DEST[15:0];
DEST[15:0] := (DEST[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] := (DEST[63:32] < 0) ? 0 : DEST[47:32];
DEST[31:16] := (DEST[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] := (DEST[95:64] < 0) ? 0 : DEST[79:64];
DEST[47:32] := (DEST[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] := (DEST[127:96] < 0) ? 0 : DEST[111:96];
DEST[63:48] := (DEST[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] := (SRC[31:0] < 0) ? 0 : SRC[15:0];
DEST[79:64] := (SRC[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] := (SRC[63:32] < 0) ? 0 : SRC[47:32];
DEST[95:80] := (SRC[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] := (SRC[95:64] < 0) ? 0 : SRC[79:64];
DEST[111:96] := (SRC[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] := (SRC[127:96] < 0) ? 0 : SRC[111:96];
DEST[127:112] := (SRC[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
DEST[MAXVL-1:128] (Unmodified)
```

PACKUSDW (VEX.128 Encoded Version)

```
TMP[15:0] := (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] := (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] := (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] := (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] := (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] := (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] := (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] := (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] := (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[79:64] := (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] := (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
DEST[95:80] := (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] := (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] := (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] := (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] := (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112];
DEST[MAXVL-1:128] := 0;
```

VPACKUSDW (VEX.256 Encoded Version)

```
TMP[15:0] := (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] := (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] := (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] := (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] := (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] := (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] := (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] := (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] := (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[79:64] := (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] := (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
DEST[95:80] := (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] := (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] := (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] := (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] := (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
TMP[143:128] := (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
DEST[143:128] := (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128] ;
TMP[159:144] := (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
DEST[159:144] := (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144] ;
TMP[175:160] := (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
DEST[175:160] := (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160] ;
TMP[191:176] := (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
DEST[191:176] := (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176] ;
TMP[207:192] := (SRC2[159:128] < 0) ? 0 : SRC2[143:128];
DEST[207:192] := (SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192] ;
TMP[223:208] := (SRC2[191:160] < 0) ? 0 : SRC2[175:160];
DEST[223:208] := (SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208] ;
TMP[239:224] := (SRC2[223:192] < 0) ? 0 : SRC2[207:192];
DEST[239:224] := (SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224] ;
TMP[255:240] := (SRC2[255:224] < 0) ? 0 : SRC2[239:224];
DEST[255:240] := (SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240] ;
DEST[MAXVL-1:256] := 0;
```

VPACKUSDW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO ((KL/2) - 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[15:0] := (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] := (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] := (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] := (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] := (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] := (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
```

```
TMP[63:48] := (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] := (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] := (TMP_SRC2[31:0] < 0) ? 0 : TMP_SRC2[15:0];
DEST[79:64] := (TMP_SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] := (TMP_SRC2[63:32] < 0) ? 0 : TMP_SRC2[47:32];
DEST[95:80] := (TMP_SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] := (TMP_SRC2[95:64] < 0) ? 0 : TMP_SRC2[79:64];
DEST[111:96] := (TMP_SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] := (TMP_SRC2[127:96] < 0) ? 0 : TMP_SRC2[111:96];
DEST[127:112] := (TMP_SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
IF VL >= 256
    TMP[143:128] := (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
    DEST[143:128] := (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128] ;
    TMP[159:144] := (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
    DEST[159:144] := (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144] ;
    TMP[175:160] := (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
    DEST[175:160] := (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160] ;
    TMP[191:176] := (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
    DEST[191:176] := (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176] ;
    TMP[207:192] := (TMP_SRC2[159:128] < 0) ? 0 : TMP_SRC2[143:128];
    DEST[207:192] := (TMP_SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192] ;
    TMP[223:208] := (TMP_SRC2[191:160] < 0) ? 0 : TMP_SRC2[175:160];
    DEST[223:208] := (TMP_SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208] ;
    TMP[239:224] := (TMP_SRC2[223:192] < 0) ? 0 : TMP_SRC2[207:192];
    DEST[239:224] := (TMP_SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224] ;
    TMP[255:240] := (TMP_SRC2[255:224] < 0) ? 0 : TMP_SRC2[239:224];
    DEST[255:240] := (TMP_SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240] ;
FI;
IF VL >= 512
    TMP[271:256] := (SRC1[287:256] < 0) ? 0 : SRC1[271:256];
    DEST[271:256] := (SRC1[287:256] > FFFFH) ? FFFFH : TMP[271:256] ;
    TMP[287:272] := (SRC1[319:288] < 0) ? 0 : SRC1[303:288];
    DEST[287:272] := (SRC1[319:288] > FFFFH) ? FFFFH : TMP[287:272] ;
    TMP[303:288] := (SRC1[351:320] < 0) ? 0 : SRC1[335:320];
    DEST[303:288] := (SRC1[351:320] > FFFFH) ? FFFFH : TMP[303:288] ;
    TMP[319:304] := (SRC1[383:352] < 0) ? 0 : SRC1[367:352];
    DEST[319:304] := (SRC1[383:352] > FFFFH) ? FFFFH : TMP[319:304] ;
    TMP[335:320] := (TMP_SRC2[287:256] < 0) ? 0 : TMP_SRC2[271:256];
    DEST[335:304] := (TMP_SRC2[287:256] > FFFFH) ? FFFFH : TMP[79:64] ;
    TMP[351:336] := (TMP_SRC2[319:288] < 0) ? 0 : TMP_SRC2[303:288];
    DEST[351:336] := (TMP_SRC2[319:288] > FFFFH) ? FFFFH : TMP[351:336] ;
    TMP[367:352] := (TMP_SRC2[351:320] < 0) ? 0 : TMP_SRC2[315:320];
    DEST[367:352] := (TMP_SRC2[351:320] > FFFFH) ? FFFFH : TMP[367:352] ;
    TMP[383:368] := (TMP_SRC2[383:352] < 0) ? 0 : TMP_SRC2[367:352];
    DEST[383:368] := (TMP_SRC2[383:352] > FFFFH) ? FFFFH : TMP[383:368] ;
    TMP[399:384] := (SRC1[415:384] < 0) ? 0 : SRC1[399:384];
    DEST[399:384] := (SRC1[415:384] > FFFFH) ? FFFFH : TMP[399:384] ;
    TMP[415:400] := (SRC1[447:416] < 0) ? 0 : SRC1[431:416];
    DEST[415:400] := (SRC1[447:416] > FFFFH) ? FFFFH : TMP[415:400] ;
    TMP[431:416] := (SRC1[479:448] < 0) ? 0 : SRC1[463:448];
    DEST[431:416] := (SRC1[479:448] > FFFFH) ? FFFFH : TMP[431:416] ;
    TMP[447:432] := (SRC1[511:480] < 0) ? 0 : SRC1[495:480];
    DEST[447:432] := (SRC1[511:480] > FFFFH) ? FFFFH : TMP[447:432] ;
    TMP[463:448] := (TMP_SRC2[415:384] < 0) ? 0 : TMP_SRC2[399:384];
```

```
    DEST[463:448] := (TMP_SRC2[415:384] > FFFFH) ? FFFFH : TMP[463:448] ;
    TMP[475:464] := (TMP_SRC2[447:416] < 0) ? 0 : TMP_SRC2[431:416];
    DEST[475:464] := (TMP_SRC2[447:416] > FFFFH) ? FFFFH : TMP[475:464] ;
    TMP[491:476] := (TMP_SRC2[479:448] < 0) ? 0 : TMP_SRC2[463:448];
    DEST[491:476] := (TMP_SRC2[479:448] > FFFFH) ? FFFFH : TMP[491:476] ;
    TMP[511:492] := (TMP_SRC2[511:480] < 0) ? 0 : TMP_SRC2[495:480];
    DEST[511:492] := (TMP_SRC2[511:480] > FFFFH) ? FFFFH : TMP[511:492] ;
FI;
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*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPACKUSDW__m512i _mm512_packus_epi32(__m512i m1, __m512i m2);
VPACKUSDW__m512i _mm512_mask_packus_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m512i _mm512_maskz_packus_epi32(__mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m256i _mm256_mask_packus_epi32(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m256i _mm256_maskz_packus_epi32(__mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m128i _mm_mask_packus_epi32(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKUSDW__m128i _mm_maskz_packus_epi32(__mmask8 k, __m128i m1, __m128i m2);
PACKUSDW__m128i _mm_packus_epi32(__m128i m1, __m128i m2);
VPACKUSDW__m256i _mm256_packus_epi32(__m256i m1, __m256i m2);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”

PACKUSWB—Pack With Unsigned Saturation

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 67 /r[1] PACKUSWB mm, mm/m64	A	V/V	MMX	Converts 4 signed word integers from mm and 4 signed word integers from mm/m64 into 8 unsigned byte integers in mm using unsigned saturation.
66 0F 67 /r PACKUSWB xmm1, xmm2/m128	A	V/V	SSE2	Converts 8 signed word integers from xmm1 and 8 signed word integers from xmm2/m128 into 16 unsigned byte integers in xmm1 using unsigned saturation.
VEX.128.66.0F.WIG 67 /r VPACKUSWB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Converts 8 signed word integers from xmm2 and 8 signed word integers from xmm3/m128 into 16 unsigned byte integers in xmm1 using unsigned saturation.
VEX.256.66.0F.WIG 67 /r VPACKUSWB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Converts 16 signed word integers from ymm2 and 16signed word integers from ymm3/m256 into 32 unsigned byte integers in ymm1 using unsigned saturation.
EVEX.128.66.0F.WIG 67 /r VPACKUSWB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Converts signed word integers from xmm2 and signed word integers from xmm3/m128 into unsigned byte integers in xmm1 using unsigned saturation under writemask k1.
EVEX.256.66.0F.WIG 67 /r VPACKUSWB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Converts signed word integers from ymm2 and signed word integers from ymm3/m256 into unsigned byte integers in ymm1 using unsigned saturation under writemask k1.
EVEX.512.66.0F.WIG 67 /r VPACKUSWB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Converts signed word integers from zmm2 and signed word integers from zmm3/m512 into unsigned byte integers in zmm1 using unsigned saturation under writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts 4, 8, 16, or 32 signed word integers from the destination operand (first operand) and 4, 8, 16, or 32 signed word integers from the source operand (second operand) into 8, 16, 32 or 64 unsigned byte integers and stores the result in the destination operand. (See Figure 1-9 for an example of the packing operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

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. 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. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand 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

PACKUSWB (With 64-bit Operands)

```
DEST[7:0] := SaturateSignedWordToUnsignedByte DEST[15:0];
DEST[15:8] := SaturateSignedWordToUnsignedByte DEST[31:16];
DEST[23:16] := SaturateSignedWordToUnsignedByte DEST[47:32];
DEST[31:24] := SaturateSignedWordToUnsignedByte DEST[63:48];
DEST[39:32] := SaturateSignedWordToUnsignedByte SRC[15:0];
DEST[47:40] := SaturateSignedWordToUnsignedByte SRC[31:16];
DEST[55:48] := SaturateSignedWordToUnsignedByte SRC[47:32];
DEST[63:56] := SaturateSignedWordToUnsignedByte SRC[63:48];
```

PACKUSWB (Legacy SSE Instruction)

```
DEST[7:0] := SaturateSignedWordToUnsignedByte (DEST[15:0]);
DEST[15:8] := SaturateSignedWordToUnsignedByte (DEST[31:16]);
DEST[23:16] := SaturateSignedWordToUnsignedByte (DEST[47:32]);
DEST[31:24] := SaturateSignedWordToUnsignedByte (DEST[63:48]);
DEST[39:32] := SaturateSignedWordToUnsignedByte (DEST[79:64]);
DEST[47:40] := SaturateSignedWordToUnsignedByte (DEST[95:80]);
DEST[55:48] := SaturateSignedWordToUnsignedByte (DEST[111:96]);
DEST[63:56] := SaturateSignedWordToUnsignedByte (DEST[127:112]);
DEST[71:64] := SaturateSignedWordToUnsignedByte (SRC[15:0]);
DEST[79:72] := SaturateSignedWordToUnsignedByte (SRC[31:16]);
DEST[87:80] := SaturateSignedWordToUnsignedByte (SRC[47:32]);
DEST[95:88] := SaturateSignedWordToUnsignedByte (SRC[63:48]);
DEST[103:96] := SaturateSignedWordToUnsignedByte (SRC[79:64]);
DEST[111:104] := SaturateSignedWordToUnsignedByte (SRC[95:80]);
DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC[111:96]);
DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC[127:112]);
```

PACKUSWB (VEX.128 Encoded Version)

```
DEST[7:0] := SaturateSignedWordToUnsignedByte (SRC1[15:0]);
DEST[15:8] := SaturateSignedWordToUnsignedByte (SRC1[31:16]);
DEST[23:16] := SaturateSignedWordToUnsignedByte (SRC1[47:32]);
DEST[31:24] := SaturateSignedWordToUnsignedByte (SRC1[63:48]);
DEST[39:32] := SaturateSignedWordToUnsignedByte (SRC1[79:64]);
DEST[47:40] := SaturateSignedWordToUnsignedByte (SRC1[95:80]);
DEST[55:48] := SaturateSignedWordToUnsignedByte (SRC1[111:96]);
DEST[63:56] := SaturateSignedWordToUnsignedByte (SRC1[127:112]);
DEST[71:64] := SaturateSignedWordToUnsignedByte (SRC2[15:0]);
```

```
    DEST[79:72] := SaturateSignedWordToUnsignedByte (SRC2[31:16]);
    DEST[87:80] := SaturateSignedWordToUnsignedByte (SRC2[47:32]);
    DEST[95:88] := SaturateSignedWordToUnsignedByte (SRC2[63:48]);
    DEST[103:96] := SaturateSignedWordToUnsignedByte (SRC2[79:64]);
    DEST[111:104] := SaturateSignedWordToUnsignedByte (SRC2[95:80]);
    DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC2[111:96]);
    DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC2[127:112]);
    DEST[MAXVL-1:128] := 0;
```

VPACKUSWB (VEX.256 Encoded Version)

```
    DEST[7:0] := SaturateSignedWordToUnsignedByte (SRC1[15:0]);
    DEST[15:8] := SaturateSignedWordToUnsignedByte (SRC1[31:16]);
    DEST[23:16] := SaturateSignedWordToUnsignedByte (SRC1[47:32]);
    DEST[31:24] := SaturateSignedWordToUnsignedByte (SRC1[63:48]);
    DEST[39:32] := SaturateSignedWordToUnsignedByte (SRC1[79:64]);
    DEST[47:40] := SaturateSignedWordToUnsignedByte (SRC1[95:80]);
    DEST[55:48] := SaturateSignedWordToUnsignedByte (SRC1[111:96]);
    DEST[63:56] := SaturateSignedWordToUnsignedByte (SRC1[127:112]);
    DEST[71:64] := SaturateSignedWordToUnsignedByte (SRC2[15:0]);
    DEST[79:72] := SaturateSignedWordToUnsignedByte (SRC2[31:16]);
    DEST[87:80] := SaturateSignedWordToUnsignedByte (SRC2[47:32]);
    DEST[95:88] := SaturateSignedWordToUnsignedByte (SRC2[63:48]);
    DEST[103:96] := SaturateSignedWordToUnsignedByte (SRC2[79:64]);
    DEST[111:104] := SaturateSignedWordToUnsignedByte (SRC2[95:80]);
    DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC2[111:96]);
    DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC2[127:112]);
    DEST[135:128] := SaturateSignedWordToUnsignedByte (SRC1[143:128]);
    DEST[143:136] := SaturateSignedWordToUnsignedByte (SRC1[159:144]);
    DEST[151:144] := SaturateSignedWordToUnsignedByte (SRC1[175:160]);
    DEST[159:152] := SaturateSignedWordToUnsignedByte (SRC1[191:176]);
    DEST[167:160] := SaturateSignedWordToUnsignedByte (SRC1[207:192]);
    DEST[175:168] := SaturateSignedWordToUnsignedByte (SRC1[223:208]);
    DEST[183:176] := SaturateSignedWordToUnsignedByte (SRC1[239:224]);
    DEST[191:184] := SaturateSignedWordToUnsignedByte (SRC1[255:240]);
    DEST[199:192] := SaturateSignedWordToUnsignedByte (SRC2[143:128]);
    DEST[207:200] := SaturateSignedWordToUnsignedByte (SRC2[159:144]);
    DEST[215:208] := SaturateSignedWordToUnsignedByte (SRC2[175:160]);
    DEST[223:216] := SaturateSignedWordToUnsignedByte (SRC2[191:176]);
    DEST[231:224] := SaturateSignedWordToUnsignedByte (SRC2[207:192]);
    DEST[239:232] := SaturateSignedWordToUnsignedByte (SRC2[223:208]);
    DEST[247:240] := SaturateSignedWordToUnsignedByte (SRC2[239:224]);
    DEST[255:248] := SaturateSignedWordToUnsignedByte (SRC2[255:240]);
```

VPACKUSWB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] := SaturateSignedWordToUnsignedByte (SRC1[15:0]);
TMP_DEST[15:8] := SaturateSignedWordToUnsignedByte (SRC1[31:16]);
TMP_DEST[23:16] := SaturateSignedWordToUnsignedByte (SRC1[47:32]);
TMP_DEST[31:24] := SaturateSignedWordToUnsignedByte (SRC1[63:48]);
TMP_DEST[39:32] := SaturateSignedWordToUnsignedByte (SRC1[79:64]);
TMP_DEST[47:40] := SaturateSignedWordToUnsignedByte (SRC1[95:80]);
TMP_DEST[55:48] := SaturateSignedWordToUnsignedByte (SRC1[111:96]);
TMP_DEST[63:56] := SaturateSignedWordToUnsignedByte (SRC1[127:112]);
TMP_DEST[71:64] := SaturateSignedWordToUnsignedByte (SRC2[15:0]);
```

```
TMP_DEST[79:72] := SaturateSignedWordToUnsignedByte (SRC2[31:16]);
TMP_DEST[87:80] := SaturateSignedWordToUnsignedByte (SRC2[47:32]);
TMP_DEST[95:88] := SaturateSignedWordToUnsignedByte (SRC2[63:48]);
TMP_DEST[103:96] := SaturateSignedWordToUnsignedByte (SRC2[79:64]);
TMP_DEST[111:104] := SaturateSignedWordToUnsignedByte (SRC2[95:80]);
TMP_DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC2[111:96]);
TMP_DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128] := SaturateSignedWordToUnsignedByte (SRC1[143:128]);
    TMP_DEST[143:136] := SaturateSignedWordToUnsignedByte (SRC1[159:144]);
    TMP_DEST[151:144] := SaturateSignedWordToUnsignedByte (SRC1[175:160]);
    TMP_DEST[159:152] := SaturateSignedWordToUnsignedByte (SRC1[191:176]);
    TMP_DEST[167:160] := SaturateSignedWordToUnsignedByte (SRC1[207:192]);
    TMP_DEST[175:168] := SaturateSignedWordToUnsignedByte (SRC1[223:208]);
    TMP_DEST[183:176] := SaturateSignedWordToUnsignedByte (SRC1[239:224]);
    TMP_DEST[191:184] := SaturateSignedWordToUnsignedByte (SRC1[255:240]);
    TMP_DEST[199:192] := SaturateSignedWordToUnsignedByte (SRC2[143:128]);
    TMP_DEST[207:200] := SaturateSignedWordToUnsignedByte (SRC2[159:144]);
    TMP_DEST[215:208] := SaturateSignedWordToUnsignedByte (SRC2[175:160]);
    TMP_DEST[223:216] := SaturateSignedWordToUnsignedByte (SRC2[191:176]);
    TMP_DEST[231:224] := SaturateSignedWordToUnsignedByte (SRC2[207:192]);
    TMP_DEST[239:232] := SaturateSignedWordToUnsignedByte (SRC2[223:208]);
    TMP_DEST[247:240] := SaturateSignedWordToUnsignedByte (SRC2[239:224]);
    TMP_DEST[255:248] := SaturateSignedWordToUnsignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] := SaturateSignedWordToUnsignedByte (SRC1[271:256]);
    TMP_DEST[271:264] := SaturateSignedWordToUnsignedByte (SRC1[287:272]);
    TMP_DEST[279:272] := SaturateSignedWordToUnsignedByte (SRC1[303:288]);
    TMP_DEST[287:280] := SaturateSignedWordToUnsignedByte (SRC1[319:304]);
    TMP_DEST[295:288] := SaturateSignedWordToUnsignedByte (SRC1[335:320]);
    TMP_DEST[303:296] := SaturateSignedWordToUnsignedByte (SRC1[351:336]);
    TMP_DEST[311:304] := SaturateSignedWordToUnsignedByte (SRC1[367:352]);
    TMP_DEST[319:312] := SaturateSignedWordToUnsignedByte (SRC1[383:368]);

    TMP_DEST[327:320] := SaturateSignedWordToUnsignedByte (SRC2[271:256]);
    TMP_DEST[335:328] := SaturateSignedWordToUnsignedByte (SRC2[287:272]);
    TMP_DEST[343:336] := SaturateSignedWordToUnsignedByte (SRC2[303:288]);
    TMP_DEST[351:344] := SaturateSignedWordToUnsignedByte (SRC2[319:304]);
    TMP_DEST[359:352] := SaturateSignedWordToUnsignedByte (SRC2[335:320]);
    TMP_DEST[367:360] := SaturateSignedWordToUnsignedByte (SRC2[351:336]);
    TMP_DEST[375:368] := SaturateSignedWordToUnsignedByte (SRC2[367:352]);
    TMP_DEST[383:376] := SaturateSignedWordToUnsignedByte (SRC2[383:368]);

    TMP_DEST[391:384] := SaturateSignedWordToUnsignedByte (SRC1[399:384]);
    TMP_DEST[399:392] := SaturateSignedWordToUnsignedByte (SRC1[415:400]);
    TMP_DEST[407:400] := SaturateSignedWordToUnsignedByte (SRC1[431:416]);
    TMP_DEST[415:408] := SaturateSignedWordToUnsignedByte (SRC1[447:432]);
    TMP_DEST[423:416] := SaturateSignedWordToUnsignedByte (SRC1[463:448]);
    TMP_DEST[431:424] := SaturateSignedWordToUnsignedByte (SRC1[479:464]);
    TMP_DEST[439:432] := SaturateSignedWordToUnsignedByte (SRC1[495:480]);
    TMP_DEST[447:440] := SaturateSignedWordToUnsignedByte (SRC1[511:496]);

    TMP_DEST[455:448] := SaturateSignedWordToUnsignedByte (SRC2[399:384]);
```

```
    TMP_DEST[463:456] := SaturateSignedWordToUnsignedByte (SRC2[415:400]);
    TMP_DEST[471:464] := SaturateSignedWordToUnsignedByte (SRC2[431:416]);
    TMP_DEST[479:472] := SaturateSignedWordToUnsignedByte (SRC2[447:432]);
    TMP_DEST[487:480] := SaturateSignedWordToUnsignedByte (SRC2[463:448]);
    TMP_DEST[495:488] := SaturateSignedWordToUnsignedByte (SRC2[479:464]);
    TMP_DEST[503:496] := SaturateSignedWordToUnsignedByte (SRC2[495:480]);
    TMP_DEST[511:504] := SaturateSignedWordToUnsignedByte (SRC2[511:496]);
FI;
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] := TMP_DEST[i+7:i]
        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] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPACKUSWB __m512i _mm512_packus_epi16(__m512i m1, __m512i m2);
VPACKUSWB __m512i _mm512_mask_packus_epi16(__m512i s, __mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB __m512i _mm512_maskz_packus_epi16(__mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB __m256i _mm256_mask_packus_epi16(__m256i s, __mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB __m256i _mm256_maskz_packus_epi16(__mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB __m128i _mm_mask_packus_epi16(__m128i s, __mmask16 k, __m128i m1, __m128i m2);
VPACKUSWB __m128i _mm_maskz_packus_epi16(__mmask16 k, __m128i m1, __m128i m2);
PACKUSWB __m64 _mm_packs_pu16(__m64 m1, __m64 m2)
(V)PACKUSWB __m128i _mm_packus_epi16(__m128i m1, __m128i m2)
VPACKUSWB __m256i _mm256_packus_epi16(__m256i m1, __m256i m2);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

PADDB/PADDW/PADDD/PADDQ—Add Packed Integers

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F FC /r[1] PADDB mm, mm/m64	A	V/V	MMX	Add packed byte integers from mm/m64 and mm.
NP 0F FD /r[1] PADDW mm, mm/m64	A	V/V	MMX	Add packed word integers from mm/m64 and mm.
NP 0F FE /r[1] PADDD mm, mm/m64	A	V/V	MMX	Add packed doubleword integers from mm/m64 and mm.
NP 0F D4 /r[1] PADDQ mm, mm/m64	A	V/V	MMX	Add packed quadword integers from mm/m64 and mm.
66 0F FC /r PADDB xmm1, xmm2/m128	A	V/V	SSE2	Add packed byte integers from xmm2/m128 and xmm1.
66 0F FD /r PADDW xmm1, xmm2/m128	A	V/V	SSE2	Add packed word integers from xmm2/m128 and xmm1.
66 0F FE /r PADDD xmm1, xmm2/m128	A	V/V	SSE2	Add packed doubleword integers from xmm2/m128 and xmm1.
66 0F D4 /r PADDQ xmm1, xmm2/m128	A	V/V	SSE2	Add packed quadword integers from xmm2/m128 and xmm1.
VEX.128.66.0F.WIG FC /r VPADDB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed byte integers from xmm2, and xmm3/m128 and store in xmm1.
VEX.128.66.0F.WIG FD /r VPADDW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed word integers from xmm2, xmm3/m128 and store in xmm1.
VEX.128.66.0F.WIG FE /r VPADDD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed doubleword integers from xmm2, xmm3/m128 and store in xmm1.
VEX.128.66.0F.WIG D4 /r VPADDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed quadword integers from xmm2, xmm3/m128 and store in xmm1.
VEX.256.66.0F.WIG FC /r VPADDB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed byte integers from ymm2, and ymm3/m256 and store in ymm1.
VEX.256.66.0F.WIG FD /r VPADDW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed word integers from ymm2, ymm3/m256 and store in ymm1.
VEX.256.66.0F.WIG FE /r VPADDD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed doubleword integers from ymm2, ymm3/m256 and store in ymm1.
VEX.256.66.0F.WIG D4 /r VPADDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed quadword integers from ymm2, ymm3/m256 and store in ymm1.
EVEX.128.66.0F.WIG FC /r VPADDB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Add packed byte integers from xmm2, and xmm3/m128 and store in xmm1 using writemask k1.
EVEX.128.66.0F.WIG FD /r VPADDW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Add packed word integers from xmm2, and xmm3/m128 and store in xmm1 using writemask k1.
EVEX.128.66.0F.W0 FE /r VPADDD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Add packed doubleword integers from xmm2, and xmm3/m128/m32bcst and store in xmm1 using writemask k1.
EVEX.128.66.0F.W1 D4 /r VPADDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Add packed quadword integers from xmm2, and xmm3/m128/m64bcst and store in xmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.WIG FC /r VPADDB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Add packed byte integers from ymm2, and ymm3/m256 and store in ymm1 using writemask k1.
EVEX.256.66.0F.WIG FD /r VPADDW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Add packed word integers from ymm2, and ymm3/m256 and store in ymm1 using writemask k1.
EVEX.256.66.0F.W0 FE /r VPADDD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Add packed doubleword integers from ymm2, ymm3/m256/m32bcst and store in ymm1 using writemask k1.
EVEX.256.66.0F.W1 D4 /r VPADDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Add packed quadword integers from ymm2, ymm3/m256/m64bcst and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG FC /r VPADDB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Add packed byte integers from zmm2, and zmm3/m512 and store in zmm1 using writemask k1.
EVEX.512.66.0F.WIG FD /r VPADDW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Add packed word integers from zmm2, and zmm3/m512 and store in zmm1 using writemask k1.
EVEX.512.66.0F.W0 FE /r VPADDD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F OR AVX10.1[2]	Add packed doubleword integers from zmm2, zmm3/m512/m32bcst and store in zmm1 using writemask k1.
EVEX.512.66.0F.W1 D4 /r VPADDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F OR AVX10.1[2]	Add packed quadword integers from zmm2, zmm3/m512/m64bcst and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD add of the packed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The PADDB and VPADDB instructions add packed byte integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW and VPADDW instructions add packed word integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand (that is, the carry is ignored).

The PADDD and VPADDD instructions add packed doubleword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand (that is, the carry is ignored).

The PADDQ and VPADDQ instructions add packed quadword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination operand (that is, the carry is ignored).

Note that the (V)PADDB, (V)PADDW, (V)PADDD and (V)PADDQ instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

EVEX encoded VPADDD/Q: 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 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPADDB/W: 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. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

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. the upper bits (MAXVL-1:256) of the destination are cleared.

VEX.128 encoded version: 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. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand 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 ZMM register destination are unmodified.

Operation

PADDB (With 64-bit Operands)

```
    DEST[7:0] := DEST[7:0] + SRC[7:0];
    (* Repeat add operation for 2nd through 7th byte *)
    DEST[63:56] := DEST[63:56] + SRC[63:56];
```

PADDW (With 64-bit Operands)

```
    DEST[15:0] := DEST[15:0] + SRC[15:0];
    (* Repeat add operation for 2nd and 3th word *)
    DEST[63:48] := DEST[63:48] + SRC[63:48];
```

PADDD (With 64-bit Operands)

```
    DEST[31:0] := DEST[31:0] + SRC[31:0];
    DEST[63:32] := DEST[63:32] + SRC[63:32];
```

PADDQ (With 64-Bit Operands)

```
    DEST[63:0] := DEST[63:0] + SRC[63:0];
```

PADDB (Legacy SSE Instruction)

```
    DEST[7:0] := DEST[7:0] + SRC[7:0];
    (* Repeat add operation for 2nd through 15th byte *)
    DEST[127:120] := DEST[127:120] + SRC[127:120];
    DEST[MAXVL-1:128] (Unmodified)
```

PADDW (Legacy SSE Instruction)

```
    DEST[15:0] := DEST[15:0] + SRC[15:0];
    (* Repeat add operation for 2nd through 7th word *)
    DEST[127:112] := DEST[127:112] + SRC[127:112];
    DEST[MAXVL-1:128] (Unmodified)
```

PADDD (Legacy SSE Instruction)

```
    DEST[31:0] := DEST[31:0] + SRC[31:0];
    (* Repeat add operation for 2nd and 3th doubleword *)
    DEST[127:96] := DEST[127:96] + SRC[127:96];
    DEST[MAXVL-1:128] (Unmodified)
```

PADDQ (Legacy SSE Instruction)

```
    DEST[63:0] := DEST[63:0] + SRC[63:0];
    DEST[127:64] := DEST[127:64] + SRC[127:64];
    DEST[MAXVL-1:128] (Unmodified)
```

VPADDB (VEX.128 Encoded Instruction)

```
    DEST[7:0] := SRC1[7:0] + SRC2[7:0];
    (* Repeat add operation for 2nd through 15th byte *)
    DEST[127:120] := SRC1[127:120] + SRC2[127:120];
    DEST[MAXVL-1:128] := 0;
```

VPADDW (VEX.128 Encoded Instruction)

```
    DEST[15:0] := SRC1[15:0] + SRC2[15:0];
    (* Repeat add operation for 2nd through 7th word *)
    DEST[127:112] := SRC1[127:112] + SRC2[127:112];
    DEST[MAXVL-1:128] := 0;
```

VPADDD (VEX.128 Encoded Instruction)

```
    DEST[31:0] := SRC1[31:0] + SRC2[31:0];
    (* Repeat add operation for 2nd and 3th doubleword *)
    DEST[127:96] := SRC1[127:96] + SRC2[127:96];
    DEST[MAXVL-1:128] := 0;
```

VPADDQ (VEX.128 Encoded Instruction)

```
    DEST[63:0] := SRC1[63:0] + SRC2[63:0];
    DEST[127:64] := SRC1[127:64] + SRC2[127:64];
    DEST[MAXVL-1:128] := 0;
```

VPADDB (VEX.256 Encoded Instruction)

```
    DEST[7:0] := SRC1[7:0] + SRC2[7:0];
    (* Repeat add operation for 2nd through 31th byte *)
    DEST[255:248] := SRC1[255:248] + SRC2[255:248];
```

VPADDW (VEX.256 Encoded Instruction)

```
    DEST[15:0] := SRC1[15:0] + SRC2[15:0];
    (* Repeat add operation for 2nd through 15th word *)
    DEST[255:240] := SRC1[255:240] + SRC2[255:240];
```

VPADDD (VEX.256 Encoded Instruction)

```
    DEST[31:0] := SRC1[31:0] + SRC2[31:0];
    (* Repeat add operation for 2nd and 7th doubleword *)
    DEST[255:224] := SRC1[255:224] + SRC2[255:224];
```

VPADDQ (VEX.256 Encoded Instruction)

```
    DEST[63:0] := SRC1[63:0] + SRC2[63:0];
    DEST[127:64] := SRC1[127:64] + SRC2[127:64];
    DEST[191:128] := SRC1[191:128] + SRC2[191:128];
    DEST[255:192] := SRC1[255:192] + SRC2[255:192];
```

VPADDB (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] := SRC1[i+7:i] + SRC2[i+7:i]
        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] := 0
```

VPADDW (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] := SRC1[i+15:i] + SRC2[i+15:i]
        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] := 0
```

VPADDD (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] + SRC2[31:0]
                ELSE DEST[i+31:i] := 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
```

VPADDQ (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] + SRC2[63:0]
                ELSE DEST[i+63:i] := 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 Equivalents

```
VPADDB__m512i _mm512_add_epi8 ( __m512i a, __m512i b)
VPADDW__m512i _mm512_add_epi16 ( __m512i a, __m512i b)
VPADDB__m512i _mm512_mask_add_epi8 ( __m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDW__m512i _mm512_mask_add_epi16 ( __m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDB__m512i _mm512_maskz_add_epi8 (__mmask64 m, __m512i a, __m512i b)
VPADDW__m512i _mm512_maskz_add_epi16 (__mmask32 m, __m512i a, __m512i b)
VPADDB__m256i _mm256_mask_add_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_mask_add_epi16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDB__m256i _mm256_maskz_add_epi8 (__mmask32 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_maskz_add_epi16 (__mmask16 m, __m256i a, __m256i b)
VPADDB__m128i _mm_mask_add_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDW__m128i _mm_mask_add_epi16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
```

```
VPADDB__m128i _mm_maskz_add_epi8 (__mmask16 m, __m128i a, __m128i b)
VPADDW__m128i _mm_maskz_add_epi16 (__mmask8 m, __m128i a, __m128i b)
VPADDD __m512i _mm512_add_epi32( __m512i a, __m512i b);
VPADDD __m512i _mm512_mask_add_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPADDD __m512i _mm512_maskz_add_epi32( __mmask16 k, __m512i a, __m512i b);
VPADDD __m256i _mm256_mask_add_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDD __m256i _mm256_maskz_add_epi32( __mmask8 k, __m256i a, __m256i b);
VPADDD __m128i _mm_mask_add_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDD __m128i _mm_maskz_add_epi32( __mmask8 k, __m128i a, __m128i b);
VPADDQ __m512i _mm512_add_epi64( __m512i a, __m512i b);
VPADDQ __m512i _mm512_mask_add_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPADDQ __m512i _mm512_maskz_add_epi64( __mmask8 k, __m512i a, __m512i b);
VPADDQ __m256i _mm256_mask_add_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDQ __m256i _mm256_maskz_add_epi64( __mmask8 k, __m256i a, __m256i b);
VPADDQ __m128i _mm_mask_add_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDQ __m128i _mm_maskz_add_epi64( __mmask8 k, __m128i a, __m128i b);
PADDB __m128i _mm_add_epi8 (__m128i a,__m128i b );
PADDW __m128i _mm_add_epi16 ( __m128i a, __m128i b);
PADDD  __m128i _mm_add_epi32 ( __m128i a, __m128i b);
PADDQ __m128i _mm_add_epi64 ( __m128i a, __m128i b);
VPADDB __m256i _mm256_add_epi8 (__m256ia,__m256i b );
VPADDW __m256i _mm256_add_epi16 ( __m256i a, __m256i b);
VPADDD __m256i _mm256_add_epi32 ( __m256i a, __m256i b);
VPADDQ __m256i _mm256_add_epi64 ( __m256i a, __m256i b);
PADDB __m64 _mm_add_pi8(__m64 m1, __m64 m2)
PADDW __m64 _mm_add_pi16(__m64 m1, __m64 m2)
PADDD __m64 _mm_add_pi32(__m64 m1, __m64 m2)
PADDQ __m64 _mm_add_si64(__m64 m1, __m64 m2)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPADDD/Q, see Table 2-49, “Type E4 Class Exception Conditions.”

EVEX-encoded VPADDB/W, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EC /r[1] PADDSB mm, mm/m64	A	V/V	MMX	Add packed signed byte integers from mm/m64 and mm and saturate the results.
66 0F EC /r PADDSB xmm1, xmm2/m128	A	V/V	SSE2	Add packed signed byte integers from xmm2/m128 and xmm1 saturate the results.
NP 0F ED /r[1] PADDSW mm, mm/m64	A	V/V	MMX	Add packed signed word integers from mm/m64 and mm and saturate the results.
66 0F ED /r PADDSW xmm1, xmm2/m128	A	V/V	SSE2	Add packed signed word integers from xmm2/m128 and xmm1 and saturate the results.
VEX.128.66.0F.WIG EC /r VPADDSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed signed byte integers from xmm3/m128 and xmm2 saturate the results.
VEX.128.66.0F.WIG ED /r VPADDSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed signed word integers from xmm3/m128 and xmm2 and saturate the results.
VEX.256.66.0F.WIG EC /r VPADDSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed signed byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.
VEX.256.66.0F.WIG ED /r VPADDSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed signed word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.
EVEX.128.66.0F.WIG EC /r VPADDSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Add packed signed byte integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG EC /r VPADDSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Add packed signed byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG EC /r VPADDSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Add packed signed byte integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.
EVEX.128.66.0F.WIG ED /r VPADDSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Add packed signed word integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG ED /r VPADDSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Add packed signed word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG ED /r VPADDSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Add packed signed word integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD add of the packed signed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

(V)PADDSB performs a SIMD add of the packed signed integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

(V)PADDSW performs a SIMD add of the packed signed word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a memory location. The destination operand is an ZMM/YMM/XMM register.

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.

VEX.128 encoded version: 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. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand 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

PADDSB (With 64-bit Operands)

```
DEST[7:0] := SaturateToSignedByte(DEST[7:0] + SRC (7:0]);
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] := SaturateToSignedByte(DEST[63:56] + SRC[63:56] );
```

PADDSB (With 128-bit Operands)

```
DEST[7:0] := SaturateToSignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToSignedByte (DEST[111:120] + SRC[127:120]);
```

VPADDSB (VEX.128 Encoded Version)

```
DEST[7:0] := SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToSignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[MAXVL-1:128] := 0
```

VPADDSB (VEX.256 Encoded Version)

```
DEST[7:0] := SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248] := SaturateToSignedByte (SRC1[255:248] + SRC2[255:248]);
```

VPADDSB (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] := SaturateToSignedByte (SRC1[i+7:i] + SRC2[i+7:i])
        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] := 0
```

```
PADDSW (with 64-bit operands)
    DEST[15:0] := SaturateToSignedWord(DEST[15:0] + SRC[15:0] );
    (* Repeat add operation for 2nd and 7th words *)
    DEST[63:48] := SaturateToSignedWord(DEST[63:48] + SRC[63:48] );
```

```
PADDSW (with 128-bit operands)
    DEST[15:0] := SaturateToSignedWord (DEST[15:0] + SRC[15:0]);
    (* Repeat add operation for 2nd through 7th words *)
    DEST[127:112] := SaturateToSignedWord (DEST[127:112] + SRC[127:112]);
```

VPADDSW (VEX.128 Encoded Version)

```
    DEST[15:0] := SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat subtract operation for 2nd through 7th words *)
    DEST[127:112] := SaturateToSignedWord (SRC1[127:112] + SRC2[127:112]);
    DEST[MAXVL-1:128] := 0
```

VPADDSW (VEX.256 Encoded Version)

```
    DEST[15:0] := SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat add operation for 2nd through 15th words *)
    DEST[255:240] := SaturateToSignedWord (SRC1[255:240] + SRC2[255:240])
```

VPADDSW (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] := SaturateToSignedWord (SRC1[i+15:i] + SRC2[i+15:i])
        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] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDSB __m64 _mm_adds_pi8(__m64 m1, __m64 m2)
(V)PADDSB __m128i _mm_adds_epi8 (__m128i a, __m128i b)
VPADDSB __m256i _mm256_adds_epi8 (__m256i a, __m256i b)
PADDSW __m64 _mm_adds_pi16(__m64 m1, __m64 m2)
(V)PADDSW __m128i _mm_adds_epi16 (__m128i a, __m128i b)
VPADDSW __m256i _mm256_adds_epi16 (__m256i a, __m256i b)
VPADDSB __m512i _mm512_adds_epi8 (__m512i a, __m512i b)
VPADDSW __m512i _mm512_adds_epi16 (__m512i a, __m512i b)
VPADDSB __m512i _mm512_mask_adds_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDSW __m512i _mm512_mask_adds_epi16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDSB __m512i _mm512_maskz_adds_epi8 (__mmask64 m, __m512i a, __m512i b)
VPADDSW __m512i _mm512_maskz_adds_epi16 (__mmask32 m, __m512i a, __m512i b)
VPADDSB __m256i _mm256_mask_adds_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDSW __m256i _mm256_mask_adds_epi16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDSB __m256i _mm256_maskz_adds_epi8 (__mmask32 m, __m256i a, __m256i b)
VPADDSW __m256i _mm256_maskz_adds_epi16 (__mmask16 m, __m256i a, __m256i b)
VPADDSB __m128i _mm_mask_adds_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDSW __m128i _mm_mask_adds_epi16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDSB __m128i _mm_maskz_adds_epi8 (__mmask16 m, __m128i a, __m128i b)
VPADDSW __m128i _mm_maskz_adds_epi16 (__mmask8 m, __m128i a, __m128i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PADDUSB/PADDUSW—Add Packed Unsigned Integers With Unsigned Saturation

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F DC /r[1] PADDUSB mm, mm/m64	A	V/V	MMX	Add packed unsigned byte integers from mm/m64 and mm and saturate the results.
66 0F DC /r PADDUSB xmm1, xmm2/m128	A	V/V	SSE2	Add packed unsigned byte integers from xmm2/m128 and xmm1 saturate the results.
NP 0F DD /r[1] PADDUSW mm, mm/m64	A	V/V	MMX	Add packed unsigned word integers from mm/m64 and mm and saturate the results.
66 0F DD /r PADDUSW xmm1, xmm2/m128	A	V/V	SSE2	Add packed unsigned word integers from xmm2/m128 to xmm1 and saturate the results.
VEX.128.660F.WIG DC /r VPADDUSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed unsigned byte integers from xmm3/m128 to xmm2 and saturate the results.
VEX.128.66.0F.WIG DD /r VPADDUSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed unsigned word integers from xmm3/m128 to xmm2 and saturate the results.
VEX.256.66.0F.WIG DC /r VPADDUSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed unsigned byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.
VEX.256.66.0F.WIG DD /r VPADDUSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed unsigned word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.
EVEX.128.66.0F.WIG DC /r VPADDUSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Add packed unsigned byte integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG DC /r VPADDUSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Add packed unsigned byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG DC /r VPADDUSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Add packed unsigned byte integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.
EVEX.128.66.0F.WIG DD /r VPADDUSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Add packed unsigned word integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG DD /r VPADDUSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Add packed unsigned word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG DD /r VPADDUSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Add packed unsigned word integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD add of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

(V)PADDUSB performs a SIMD add of the packed unsigned integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

(V)PADDUSW performs a SIMD add of the packed unsigned word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination is an ZMM/YMM/XMM register.

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.

VEX.128 encoded version: 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. The upper bits (MAXVL-1:128) of the corresponding destination register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand 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

PADDUSB (With 64-bit Operands)

```
    DEST[7:0] := SaturateToUnsignedByte(DEST[7:0] + SRC (7:0] );
    (* Repeat add operation for 2nd through 7th bytes *)
    DEST[63:56] := SaturateToUnsignedByte(DEST[63:56] + SRC[63:56]
```

PADDUSB (With 128-bit Operands)

```
    DEST[7:0] := SaturateToUnsignedByte (DEST[7:0] + SRC[7:0]);
    (* Repeat add operation for 2nd through 14th bytes *)
    DEST[127:120] := SaturateToUnSignedByte (DEST[127:120] + SRC[127:120]);
```

VPADDUSB (VEX.128 Encoded Version)

```
    DEST[7:0] := SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
    (* Repeat subtract operation for 2nd through 14th bytes *)
    DEST[127:120] := SaturateToUnsignedByte (SRC1[111:120] + SRC2[127:120]);
    DEST[MAXVL-1:128] := 0
```

VPADDUSB (VEX.256 Encoded Version)

```
    DEST[7:0] := SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
    (* Repeat add operation for 2nd through 31st bytes *)
    DEST[255:248] := SaturateToUnsignedByte (SRC1[255:248] + SRC2[255:248]);
```

PADDUSW (With 64-bit Operands)

```
    DEST[15:0] := SaturateToUnsignedWord(DEST[15:0] + SRC[15:0] );
    (* Repeat add operation for 2nd and 3rd words *)
    DEST[63:48] := SaturateToUnsignedWord(DEST[63:48] + SRC[63:48] );
```

PADDUSW (With 128-bit Operands)

```
    DEST[15:0] := SaturateToUnsignedWord (DEST[15:0] + SRC[15:0]);
    (* Repeat add operation for 2nd through 7th words *)
    DEST[127:112] := SaturateToUnSignedWord (DEST[127:112] + SRC[127:112]);
```

VPADDUSW (VEX.128 Encoded Version)

```
    DEST[15:0] := SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat subtract operation for 2nd through 7th words *)
    DEST[127:112] := SaturateToUnsignedWord (SRC1[127:112] + SRC2[127:112]);
    DEST[MAXVL-1:128] := 0
```

VPADDUSW (VEX.256 Encoded Version)

```
    DEST[15:0] := SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat add operation for 2nd through 15th words *)
    DEST[255:240] := SaturateToUnsignedWord (SRC1[255:240] + SRC2[255:240])
```

VPADDUSB (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] := SaturateToUnsignedByte (SRC1[i+7:i] + SRC2[i+7:i])
        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] := 0
```

VPADDUSW (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] := SaturateToUnsignedWord (SRC1[i+15:i] + SRC2[i+15:i])
        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] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDUSB __m64 _mm_adds_pu8(__m64 m1, __m64 m2)
PADDUSW __m64 _mm_adds_pu16(__m64 m1, __m64 m2)
(V)PADDUSB __m128i _mm_adds_epu8 (__m128i a, __m128i b)
(V)PADDUSW __m128i _mm_adds_epu16 (__m128i a, __m128i b)
VPADDUSB __m256i _mm256_adds_epu8 (__m256i a, __m256i b)
VPADDUSW __m256i _mm256_adds_epu16 (__m256i a, __m256i b)
VPADDUSB __m512i _mm512_adds_epu8 (__m512i a, __m512i b)
VPADDUSW __m512i _mm512_adds_epu16 (__m512i a, __m512i b)
VPADDUSB __m512i _mm512_mask_adds_epu8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDUSW __m512i _mm512_mask_adds_epu16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDUSB __m512i _mm512_maskz_adds_epu8 (__mmask64 m, __m512i a, __m512i b)
VPADDUSW __m512i _mm512_maskz_adds_epu16 (__mmask32 m, __m512i a, __m512i b)
VPADDUSB __m256i _mm256_mask_adds_epu8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDUSW __m256i _mm256_mask_adds_epu16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDUSB __m256i _mm256_maskz_adds_epu8 (__mmask32 m, __m256i a, __m256i b)
VPADDUSW __m256i _mm256_maskz_adds_epu16 (__mmask16 m, __m256i a, __m256i b)
VPADDUSB __m128i _mm_mask_adds_epu8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDUSW __m128i _mm_mask_adds_epu16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDUSB __m128i _mm_maskz_adds_epu8 (__mmask16 m, __m128i a, __m128i b)
VPADDUSW __m128i _mm_maskz_adds_epu16 (__mmask8 m, __m128i a, __m128i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

PALIGNR—Packed Align Right

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 3A 0F /r ib[1] PALIGNR mm1, mm2/m64, imm8	A	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in imm8 into mm1.
66 0F 3A 0F /r ib PALIGNR xmm1, xmm2/m128, imm8	A	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in imm8 into xmm1.
VEX.128.66.0F3A.WIG 0F /r ib VPALIGNR xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Concatenate xmm2 and xmm3/m128, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.
VEX.256.66.0F3A.WIG 0F /r ib VPALIGNR ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX2	Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and two 16-byte results are stored in ymm1.
EVEX.128.66.0F3A.WIG 0F /r ib VPALIGNR xmm1 {k1}{z}, xmm2, xmm3/m128, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Concatenate xmm2 and xmm3/m128 into a 32-byte intermediate result, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.
EVEX.256.66.0F3A.WIG 0F /r ib VPALIGNR ymm1 {k1}{z}, ymm2, ymm3/m256, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and two 16-byte results are stored in ymm1.
EVEX.512.66.0F3A.WIG 0F /r ib VPALIGNR zmm1 {k1}{z}, zmm2, zmm3/m512, imm8	C	V/V	AVX512BW OR AVX10.1[1]	Concatenate pairs of 16 bytes in zmm2 and zmm3/m512 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and four 16-byte results are stored in zmm1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(V)PALIGNR concatenates the destination operand (the first operand) and the source operand (the second operand) into an intermediate composite, shifts the composite at byte granularity to the right by a constant immediate, and extracts the right-aligned result into the destination. The first and the second operands can be an MMX, XMM or a YMM register. The immediate value is considered unsigned. Immediate shift counts larger than the 2L (i.e., 32 for 128-bit operands, or 16 for 64-bit operands) produce a zero result. Both operands can be MMX registers, XMM registers or YMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded by VEX/EVEX prefix, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

EVEX.512 encoded version: The first source operand is a ZMM register and contains four 16-byte blocks. The second source operand is a ZMM register or a 512-bit memory location containing four 16-byte block. The destination operand is a ZMM register and contain four 16-byte results. The imm8[7:0] is the common shift count

used for each of the four successive 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand and so on for the blocks in the middle.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register and contains two 16-byte blocks. The second source operand is a YMM register or a 256-bit memory location containing two 16-byte block. The destination operand is a YMM register and contain two 16-byte results. The imm8[7:0] is the common shift count used for the two lower 16-byte block sources and the two upper 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand. 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. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Concatenation is done with 128-bit data in the first and second source operand for both 128-bit and 256-bit instructions. The high 128-bits of the intermediate composite 256-bit result came from the 128-bit data from the first source operand; the low 128-bits of the intermediate result came from the 128-bit data of the second source operand.

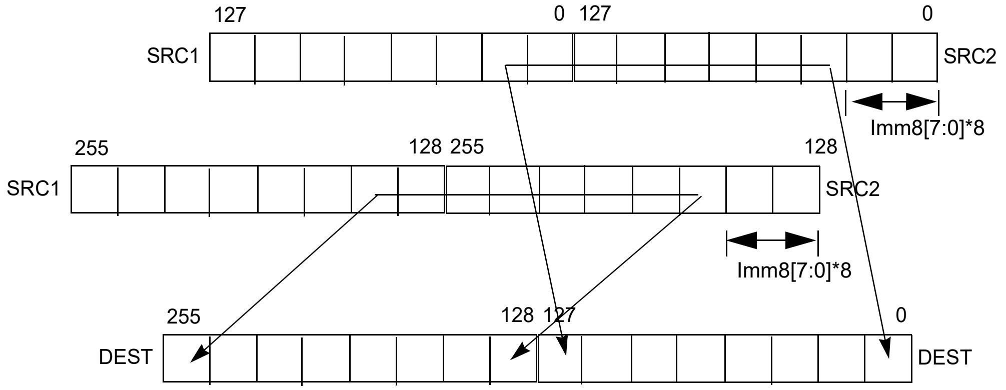


Figure 1-10. 256-bit VPALIGN Instruction Operation

Operation

PALIGNR (With 64-bit Operands)

```
temp1[127:0] = CONCATENATE(DEST,SRC)>>(imm8*8)
DEST[63:0] = temp1[63:0]
```

PALIGNR (With 128-bit Operands)

```
temp1[255:0] := ((DEST[127:0] << 128) OR SRC[127:0])>>(imm8*8);
DEST[127:0] := temp1[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

VPALIGNR (VEX.128 Encoded Version)

```
temp1[255:0] := ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8*8);
DEST[127:0] := temp1[127:0]
DEST[MAXVL-1:128] := 0
```

VPALIGNR (VEX.256 Encoded Version)

```
temp1[255:0] := ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8[7:0]*8);
DEST[127:0] := temp1[127:0]
temp1[255:0] := ((SRC1[255:128] << 128) OR SRC2[255:128])>>(imm8[7:0]*8);
DEST[MAXVL-1:128] := temp1[127:0]
```

VPALIGNR (EVEX Encoded Versions)

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

FOR l := 0 TO VL-1 with increments of 128
    temp1[255:0] := ((SRC1[l+127:l] << 128) OR SRC2[l+127:l])>>(imm8[7:0]*8);
    TMP_DEST[l+127:l] := temp1[127:0]
ENDFOR;

FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TMP_DEST[i+7:i]
        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] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PALIGNR __m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)
(V)PALIGNR __m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)
VPALIGNR __m256i _mm256_alignr_epi8 (__m256i a, __m256i b, const int n)
VPALIGNR __m512i _mm512_alignr_epi8 (__m512i a, __m512i b, const int n)
VPALIGNR __m512i _mm512_mask_alignr_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b, const int n)
VPALIGNR __m512i _mm512_maskz_alignr_epi8 (__mmask64 m, __m512i a, __m512i b, const int n)
VPALIGNR __m256i _mm256_mask_alignr_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b, const int n)
VPALIGNR __m256i _mm256_maskz_alignr_epi8 (__mmask32 m, __m256i a, __m256i b, const int n)
VPALIGNR __m128i _mm_mask_alignr_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b, const int n)
VPALIGNR __m128i _mm_maskz_alignr_epi8 (__mmask16 m, __m128i a, __m128i b, const int n)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

PAND—Logical AND

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DB /r[1] PAND mm, mm/m64	A	V/V	MMX	Bitwise AND mm/m64 and mm.
66 0F DB /r PAND xmm1, xmm2/m128	A	V/V	SSE2	Bitwise AND of xmm2/m128 and xmm1.
VEX.128.66.0F.WIG DB /r VPAND xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise AND of xmm3/m128 and xmm.
VEX.256.66.0F.WIG DB /r VPAND ymm1, ymm2, ymm3/.m256	B	V/V	AVX2	Bitwise AND of ymm2, and ymm3/m256 and store result in ymm1.
EVEX.128.66.0F.W0 DB /r VPANDD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W0 DB /r VPANDD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W0 DB /r VPANDD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1[2]	Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.128.66.0F.W1 DB /r VPANDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W1 DB /r VPANDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W1 DB /r VPANDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1[2]	Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a bitwise logical AND operation on the first source operand and second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand 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 ZMM register destination are unmodified.

EVEX encoded versions: 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/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.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. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.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. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Operation

PAND (64-bit Operand)

```
DEST := DEST AND SRC
```

PAND (128-bit Legacy SSE Version)

```
DEST := DEST AND SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPAND (VEX.128 Encoded Version)

```
DEST := SRC1 AND SRC2
DEST[MAXVL-1:128] := 0
```

VPAND (VEX.256 Encoded Instruction)

```
DEST[255:0] := (SRC1[255:0] AND SRC2[255:0])
DEST[MAXVL-1:256] := 0
```

VPANDD (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 AND SRC2[31:0]
                ELSE DEST[i+31:i] := SRC1[i+31:i] BITWISE AND 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
```

VPANDQ (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 AND SRC2[63:0]
                ELSE DEST[i+63:i] := SRC1[i+63:i] BITWISE AND 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 Equivalents

```
VPANDD __m512i _mm512_and_epi32( __m512i a, __m512i b);
VPANDD __m512i _mm512_mask_and_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDD __m512i _mm512_maskz_and_epi32( __mmask16 k, __m512i a, __m512i b);
VPANDQ __m512i _mm512_and_epi64( __m512i a, __m512i b);
VPANDQ __m512i _mm512_mask_and_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDQ __m512i _mm512_maskz_and_epi64( __mmask8 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_mask_and_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_and_epi32( __mmask8 k, __m256i a, __m256i b);
VPANDND __m128i _mm_mask_and_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i _mm_maskz_and_epi32( __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m256i _mm256_mask_and_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_maskz_and_epi64( __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m128i _mm_mask_and_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m128i _mm_maskz_and_epi64( __mmask8 k, __m128i a, __m128i b);
PAND __m64 _mm_and_si64 (__m64 m1, __m64 m2)
(V)PAND __m128i _mm_and_si128 ( __m128i a, __m128i b)
VPAND  __m256i _mm256_and_si256 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PANDN—Logical AND NOT

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F DF /r[1] PANDN mm, mm/m64	A	V/V	MMX	Bitwise AND NOT of mm/m64 and mm.
66 0F DF /r PANDN xmm1, xmm2/m128	A	V/V	SSE2	Bitwise AND NOT of xmm2/m128 and xmm1.
VEX.128.66.0F.WIG DF /r VPANDN xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise AND NOT of xmm3/m128 and xmm2.
VEX.256.66.0F.WIG DF /r VPANDN ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Bitwise AND NOT of ymm2, and ymm3/m256 and store result in ymm1.
EVEX.128.66.0F.W0 DF /r VPANDND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise AND NOT of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W0 DF /r VPANDND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise AND NOT of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W0 DF /r VPANDND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1[2]	Bitwise AND NOT of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.128.66.0F.W1 DF /r VPANDNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise AND NOT of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W1 DF /r VPANDNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise AND NOT of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W1 DF /r VPANDNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1[2]	Bitwise AND NOT of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a bitwise logical NOT operation on the first source operand, then performs bitwise AND with second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand 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 ZMM register destination are unmodified.

EVEX encoded versions: 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/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.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. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.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. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Operation

PANDN (64-bit Operand)

```
DEST := NOT(DEST) AND SRC
```

PANDN (128-bit Legacy SSE Version)

```
DEST := NOT(DEST) AND SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPANDN (VEX.128 Encoded Version)

```
DEST := NOT(SRC1) AND SRC2
DEST[MAXVL-1:128] := 0
```

VPANDN (VEX.256 Encoded Instruction)

```
DEST[255:0] := ((NOT SRC1[255:0]) AND SRC2[255:0])
DEST[MAXVL-1:256] := 0
```

VPANDND (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] := ((NOT SRC1[i+31:i]) AND SRC2[31:0])
               ELSE DEST[i+31:i] := ((NOT SRC1[i+31:i]) AND 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
```

VPANDNQ (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] := ((NOT SRC1[i+63:i]) AND SRC2[63:0])
                ELSE DEST[i+63:i] := ((NOT SRC1[i+63:i]) AND 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 Equivalents

```
VPANDND __m512i _mm512_andnot_epi32( __m512i a, __m512i b);
VPANDND __m512i _mm512_mask_andnot_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDND __m512i _mm512_maskz_andnot_epi32( __mmask16 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_mask_andnot_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_andnot_epi32( __mmask8 k, __m256i a, __m256i b);
VPANDND __m128i _mm_mask_andnot_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i _mm_maskz_andnot_epi32( __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m512i _mm512_andnot_epi64( __m512i a, __m512i b);
VPANDNQ __m512i _mm512_mask_andnot_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDNQ __m512i _mm512_maskz_andnot_epi64( __mmask8 k, __m512i a, __m512i b);
VPANDNQ __m256i _mm256_mask_andnot_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_maskz_andnot_epi64( __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m128i _mm_mask_andnot_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m128i _mm_maskz_andnot_epi64( __mmask8 k, __m128i a, __m128i b);
PANDN __m64 _mm_andnot_si64 (__m64 m1, __m64 m2)
(V)PANDN __m128i _mm_andnot_si128 ( __m128i a, __m128i b)
VPANDN __m256i _mm256_andnot_si256 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded instruction, see Table 2-49, "Type E4 Class Exception Conditions."

PAVGB/PAVGW—Average Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E0 /r[1] PAVGB mm1, mm2/m64	A	V/V	SSE	Average packed unsigned byte integers from mm2/m64 and mm1 with rounding.
66 0F E0, /r PAVGB xmm1, xmm2/m128	A	V/V	SSE2	Average packed unsigned byte integers from xmm2/m128 and xmm1 with rounding.
NP 0F E3 /r[1] PAVGW mm1, mm2/m64	A	V/V	SSE	Average packed unsigned word integers from mm2/m64 and mm1 with rounding.
66 0F E3 /r PAVGW xmm1, xmm2/m128	A	V/V	SSE2	Average packed unsigned word integers from xmm2/m128 and xmm1 with rounding.
VEX.128.66.0F.WIG E0 /r VPAVGB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Average packed unsigned byte integers from xmm3/m128 and xmm2 with rounding.
VEX.128.66.0F.WIG E3 /r VPAVGW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Average packed unsigned word integers from xmm3/m128 and xmm2 with rounding.
VEX.256.66.0F.WIG E0 /r VPAVGB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Average packed unsigned byte integers from ymm2, and ymm3/m256 with rounding and store to ymm1.
VEX.256.66.0F.WIG E3 /r VPAVGW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Average packed unsigned word integers from ymm2, ymm3/m256 with rounding to ymm1.
EVEX.128.66.0F.WIG E0 /r VPAVGB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Average packed unsigned byte integers from xmm2, and xmm3/m128 with rounding and store to xmm1 under writemask k1.
EVEX.256.66.0F.WIG E0 /r VPAVGB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Average packed unsigned byte integers from ymm2, and ymm3/m256 with rounding and store to ymm1 under writemask k1.
EVEX.512.66.0F.WIG E0 /r VPAVGB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Average packed unsigned byte integers from zmm2, and zmm3/m512 with rounding and store to zmm1 under writemask k1.
EVEX.128.66.0F.WIG E3 /r VPAVGW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Average packed unsigned word integers from xmm2, xmm3/m128 with rounding to xmm1 under writemask k1.
EVEX.256.66.0F.WIG E3 /r VPAVGW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Average packed unsigned word integers from ymm2, ymm3/m256 with rounding to ymm1 under writemask k1.
EVEX.512.66.0F.WIG E3 /r VPAVGW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Average packed unsigned word integers from zmm2, zmm3/m512 with rounding to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD average of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position.

The (V)PAVGB instruction operates on packed unsigned bytes and the (V)PAVGW instruction operates on packed unsigned words.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand 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.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

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.

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. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

Operation

PAVGB (With 64-bit Operands)

```
    DEST[7:0] := (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
    (* Repeat operation performed for bytes 2 through 6 *)
    DEST[63:56] := (SRC[63:56] + DEST[63:56] + 1) >> 1;
```

PAVGW (With 64-bit Operands)

```
    DEST[15:0] := (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
    (* Repeat operation performed for words 2 and 3 *)
    DEST[63:48] := (SRC[63:48] + DEST[63:48] + 1) >> 1;
```

PAVGB (With 128-bit Operands)

```
    DEST[7:0] := (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
    (* Repeat operation performed for bytes 2 through 14 *)
    DEST[127:120] := (SRC[127:120] + DEST[127:120] + 1) >> 1;
```

PAVGW (With 128-bit Operands)

```
    DEST[15:0] := (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
    (* Repeat operation performed for words 2 through 6 *)
    DEST[127:112] := (SRC[127:112] + DEST[127:112] + 1) >> 1;
```

VPAVGB (VEX.128 Encoded Version)

```
    DEST[7:0] := (SRC1[7:0] + SRC2[7:0] + 1) >> 1;
    (* Repeat operation performed for bytes 2 through 15 *)
    DEST[127:120] := (SRC1[127:120] + SRC2[127:120] + 1) >> 1
    DEST[MAXVL-1:128] := 0
```

VPAVGW (VEX.128 Encoded Version)

```
    DEST[15:0] := (SRC1[15:0] + SRC2[15:0] + 1) >> 1;
    (* Repeat operation performed for 16-bit words 2 through 7 *)
    DEST[127:112] := (SRC1[127:112] + SRC2[127:112] + 1) >> 1
    DEST[MAXVL-1:128] := 0
```

VPAVGB (VEX.256 Encoded Instruction)

```
    DEST[7:0] := (SRC1[7:0] + SRC2[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
    (* Repeat operation performed for bytes 2 through 31)
    DEST[255:248] := (SRC1[255:248] + SRC2[255:248] + 1) >> 1;
```

VPAVGW (VEX.256 Encoded Instruction)

```
    DEST[15:0] := (SRC1[15:0] + SRC2[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
    (* Repeat operation performed for words 2 through 15)
    DEST[255:14]) := (SRC1[255:240] + SRC2[255:240] + 1) >> 1;
VPAVGB (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] := (SRC1[i+7:i] + SRC2[i+7:i] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
        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] := 0
```

VPAVGW (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] := (SRC1[i+15:i] + SRC2[i+15:i] + 1) >> 1
                                            ; (* Temp sum before shifting is 17 bits *)
        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] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPAVGB __m512i _mm512_avg_epu8(__m512i a, __m512i b);
VPAVGW __m512i _mm512_avg_epu16(__m512i a, __m512i b);
VPAVGB __m512i _mm512_mask_avg_epu8(__m512i s, __mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_mask_avg_epu16(__m512i s, __mmask32 m, __m512i a, __m512i b);
VPAVGB __m512i _mm512_maskz_avg_epu8(__mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_maskz_avg_epu16(__mmask32 m, __m512i a, __m512i b);
VPAVGB __m256i _mm256_mask_avg_epu8(__m256i s, __mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_mask_avg_epu16(__m256i s, __mmask16 m, __m256i a, __m256i b);
VPAVGB __m256i _mm256_maskz_avg_epu8(__mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_maskz_avg_epu16(__mmask16 m, __m256i a, __m256i b);
VPAVGB __m128i _mm_mask_avg_epu8(__m128i s, __mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_mask_avg_epu16(__m128i s, __mmask8 m, __m128i a, __m128i b);
VPAVGB __m128i _mm_maskz_avg_epu8(__mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_maskz_avg_epu16(__mmask8 m, __m128i a, __m128i b);
PAVGB __m64 _mm_avg_pu8 (__m64 a, __m64 b)
PAVGW __m64 _mm_avg_pu16 (__m64 a, __m64 b)
(V)PAVGB __m128i _mm_avg_epu8 (__m128i a, __m128i b)
(V)PAVGW __m128i _mm_avg_epu16 (__m128i a, __m128i b)
VPAVGB __m256i _mm256_avg_epu8 (__m256i a, __m256i b)
VPAVGW __m256i _mm256_avg_epu16 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PCLMULQDQ—Carry-Less Multiplication Quadword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 44 /r ib PCLMULQDQ xmm1, xmm2/m128, imm8	A	V/V	PCLMULQDQ	Carry-less multiplication of one quadword of xmm1 by one quadword of xmm2/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm1 and xmm2/m128 should be used.
VEX.128.66.0F3A.WIG 44 /r ib VPCLMULQDQ xmm1, xmm2, xmm3/m128, imm8	B	V/V	PCLMULQDQ AVX	Carry-less multiplication of one quadword of xmm2 by one quadword of xmm3/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm2 and xmm3/m128 should be used.
VEX.256.66.0F3A.WIG 44 /r /ib VPCLMULQDQ ymm1, ymm2, ymm3/m256, imm8	B	V/V	VPCLMULQDQ	Carry-less multiplication of one quadword of ymm2 by one quadword of ymm3/m256, stores the 128-bit result in ymm1. The immediate is used to determine which quadwords of ymm2 and ymm3/m256 should be used.
EVEX.128.66.0F3A.WIG 44 /r /ib VPCLMULQDQ xmm1, xmm2, xmm3/m128, imm8	C	V/V	VPCLMULQDQ (AVX512VL OR AVX10.1[1])	Carry-less multiplication of one quadword of xmm2 by one quadword of xmm3/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm2 and xmm3/m128 should be used.
EVEX.256.66.0F3A.WIG 44 /r /ib VPCLMULQDQ ymm1, ymm2, ymm3/m256, imm8	C	V/V	VPCLMULQDQ (AVX512VL OR AVX10.1[1])	Carry-less multiplication of one quadword of ymm2 by one quadword of ymm3/m256, stores the 128-bit result in ymm1. The immediate is used to determine which quadwords of ymm2 and ymm3/m256 should be used.
EVEX.512.66.0F3A.WIG 44 /r /ib VPCLMULQDQ zmm1, zmm2, zmm3/m512, imm8	C	V/V	VPCLMULQDQ (AVX512F OR AVX10.1[1])	Carry-less multiplication of one quadword of zmm2 by one quadword of zmm3/m512, stores the 128-bit result in zmm1. The immediate is used to determine which quadwords of zmm2 and zmm3/m512 should be used.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

Performs a carry-less multiplication of two quadwords, selected from the first source and second source operand according to the value of the immediate byte. Bits 4 and 0 are used to select which 64-bit half of each operand to use according to Table 1-5, other bits of the immediate byte are ignored.

The EVEX encoded form of this instruction does not support memory fault suppression.

Table 1-5. PCLMULQDQ Quadword Selection of Immediate Byte

Imm[4]	Imm[0]	PCLMULQDQ Operation
0	0	CL_MUL(SRC2[1][63:0], SRC1[63:0])
0	1	CL_MUL(SRC2[63:0], SRC1[127:64])
1	0	CL_MUL(SRC2[127:64], SRC1[63:0])
1	1	CL_MUL(SRC2[127:64], SRC1[127:64])

NOTES:

1. SRC2 denotes the second source operand, which can be a register or memory; SRC1 denotes the first source and destination operand.

The first source operand and the destination operand are the same and must be a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. Bits (VL_MAX-1:128) of the corresponding YMM destination register remain unchanged.

Compilers and assemblers may implement the following pseudo-op syntax to simplify programming and emit the required encoding for imm8.

Table 1-6. Pseudo-Op and PCLMULQDQ Implementation

Pseudo-Op	Imm8 Encoding
PCLMULLQLQDQ xmm1, xmm2	**0000_0000B**
PCLMULHQLQDQ xmm1, xmm2	**0000_0001B**
PCLMULLQHQDQ xmm1, xmm2	**0001_0000B**
PCLMULHQHQDQ xmm1, xmm2	**0001_0001B**

Operation

```
define PCLMUL128(X,Y):                    // helper function
    FOR i := 0 to 63:
        TMP [ i ] := X[ 0 ] and Y[ i ]
        FOR j := 1 to i:
            TMP [ i ] := TMP [ i ] xor (X[ j ] and Y[ i - j ])
        DEST[ i ] := TMP[ i ]
    FOR i := 64 to 126:
        TMP [ i ] := 0
        FOR j := i - 63 to 63:
            TMP [ i ] := TMP [ i ] xor (X[ j ] and Y[ i - j ])
        DEST[ i ] := TMP[ i ]
    DEST[127] := 0;
    RETURN DEST                           // 128b vector
```

PCLMULQDQ (SSE Version)

```
IF imm8[0] = 0:
    TEMP1 := SRC1.qword[0]
ELSE:
    TEMP1 := SRC1.qword[1]
IF imm8[4] = 0:
    TEMP2 := SRC2.qword[0]
ELSE:
    TEMP2 := SRC2.qword[1]
DEST[127:0] := PCLMUL128(TEMP1, TEMP2)
DEST[MAXVL-1:128] (Unmodified)
```

VPCLMULQDQ (128b and 256b VEX Encoded Versions)

```
(KL,VL) = (1,128), (2,256)
FOR i= 0 to KL-1:
    IF imm8[0] = 0:
        TEMP1 := SRC1.xmm[i].qword[0]
    ELSE:
        TEMP1 := SRC1.xmm[i].qword[1]
    IF imm8[4] = 0:
        TEMP2 := SRC2.xmm[i].qword[0]
    ELSE:
        TEMP2 := SRC2.xmm[i].qword[1]
    DEST.xmm[i] := PCLMUL128(TEMP1, TEMP2)
DEST[MAXVL-1:VL] := 0
```

VPCLMULQDQ (EVEX Encoded Version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    IF imm8[0] = 0:
        TEMP1 := SRC1.xmm[i].qword[0]
    ELSE:
        TEMP1 := SRC1.xmm[i].qword[1]
    IF imm8[4] = 0:
        TEMP2 := SRC2.xmm[i].qword[0]
    ELSE:
        TEMP2 := SRC2.xmm[i].qword[1]
    DEST.xmm[i] := PCLMUL128(TEMP1, TEMP2)
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PCLMULQDQ __m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int)
VPCLMULQDQ __m256i _mm256_clmulepi64_epi128(__m256i, __m256i, const int);
VPCLMULQDQ __m512i _mm512_clmulepi64_epi128(__m512i, __m512i, const int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions,” additionally:

#UD	If VEX.L = 1.

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

PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 74 /r[1] PCMPEQB mm, mm/m64	A	V/V	MMX	Compare packed bytes in mm/m64 and mm for equality.
66 0F 74 /r PCMPEQB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed bytes in xmm2/m128 and xmm1 for equality.
NP 0F 75 /r[1] PCMPEQW mm, mm/m64	A	V/V	MMX	Compare packed words in mm/m64 and mm for equality.
66 0F 75 /r PCMPEQW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed words in xmm2/m128 and xmm1 for equality.
NP 0F 76 /r[1] PCMPEQD mm, mm/m64	A	V/V	MMX	Compare packed doublewords in mm/m64 and mm for equality.
66 0F 76 /r PCMPEQD xmm1, xmm2/m128	A	V/V	SSE2	Compare packed doublewords in xmm2/m128 and xmm1 for equality.
VEX.128.66.0F.WIG 74 /r VPCMPEQB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed bytes in xmm3/m128 and xmm2 for equality.
VEX.128.66.0F.WIG 75 /r VPCMPEQW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed words in xmm3/m128 and xmm2 for equality.
VEX.128.66.0F.WIG 76 /r VPCMPEQD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed doublewords in xmm3/m128 and xmm2 for equality.
VEX.256.66.0F.WIG 74 /r VPCMPEQB ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed bytes in ymm3/m256 and ymm2 for equality.
VEX.256.66.0F.WIG 75 /r VPCMPEQW ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed words in ymm3/m256 and ymm2 for equality.
VEX.256.66.0F.WIG 76 /r VPCMPEQD ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed doublewords in ymm3/m256 and ymm2 for equality.
EVEX.128.66.0F.W0 76 /r VPCMPEQD k1 {k2}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Compare Equal between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.W0 76 /r VPCMPEQD k1 {k2}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Compare Equal between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.W0 76 /r VPCMPEQD k1 {k2}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1[2]	Compare Equal between int32 vectors in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask k2.
EVEX.128.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, xmm2, xmm3 /m128	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed bytes in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, ymm2, ymm3 /m256	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed bytes in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, zmm2, zmm3 /m512	D	V/V	AVX512BW OR AVX10.1[2]	Compare packed bytes in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.128.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, xmm2, xmm3 /m128	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed words in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, ymm2, ymm3 /m256	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed words in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, zmm2, zmm3 /m512	D	V/V	AVX512BW OR AVX10.1[2]	Compare packed words in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare for equality of the packed bytes, words, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The (V)PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the (V)PCMPEQW instruction compares the corresponding words in the destination and source operands; and the (V)PCMPEQD instruction compares the corresponding doublewords in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE 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 destination register remain 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.

EVEX encoded VPCMPEQD: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPEQB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPEQB (With 64-bit Operands)

```
    IF DEST[7:0] = SRC[7:0]
        THEN DEST[7:0) := FFH;
        ELSE DEST[7:0] := 0; FI;
    (* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
    IF DEST[63:56] = SRC[63:56]
        THEN DEST[63:56] := FFH;
        ELSE DEST[63:56] := 0; FI;
```

COMPARE_BYTES_EQUAL (SRC1, SRC2)

```
    IF SRC1[7:0] = SRC2[7:0]
    THEN DEST[7:0] := FFH;
    ELSE DEST[7:0] := 0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
    IF SRC1[127:120] = SRC2[127:120]
    THEN DEST[127:120] := FFH;
    ELSE DEST[127:120] := 0; FI;
```

COMPARE_WORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[15:0] = SRC2[15:0]
    THEN DEST[15:0] := FFFFH;
    ELSE DEST[15:0] := 0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
    IF SRC1[127:112] = SRC2[127:112]
    THEN DEST[127:112] := FFFFH;
    ELSE DEST[127:112] := 0; FI;
```

COMPARE_DWORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[31:0] = SRC2[31:0]
    THEN DEST[31:0] := FFFFFFFFH;
    ELSE DEST[31:0] := 0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
    IF SRC1[127:96] = SRC2[127:96]
    THEN DEST[127:96] := FFFFFFFFH;
    ELSE DEST[127:96] := 0; FI;
```

PCMPEQB (With 128-bit Operands)

```
DEST[127:0] := COMPARE_BYTES_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPEQB (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] := 0
```

VPCMPEQB (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_BYTES_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPEQB (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
            /* signed comparison */
            CMP := SRC1[i+7:i] == 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
```

PCMPEQW (With 64-bit Operands)

```
    IF DEST[15:0] = SRC[15:0]
        THEN DEST[15:0] := FFFFH;
        ELSE DEST[15:0] := 0; FI;
    (* Continue comparison of 2nd and 3rd words in DEST and SRC *)
    IF DEST[63:48] = SRC[63:48]
        THEN DEST[63:48] := FFFFH;
        ELSE DEST[63:48] := 0; FI;
```

PCMPEQW (With 128-bit Operands)

```
DEST[127:0] := COMPARE_WORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPEQW (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] := 0
```

VPCMPEQW (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_WORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPEQW (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
            /* signed comparison */
            CMP := SRC1[i+15:i] == SRC2[i+15: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
```

PCMPEQD (With 64-bit Operands)

```
    IF DEST[31:0] = SRC[31:0]
        THEN DEST[31:0] := FFFFFFFFH;
        ELSE DEST[31:0] := 0; FI;
    IF DEST[63:32] = SRC[63:32]
        THEN DEST[63:32] := FFFFFFFFH;
        ELSE DEST[63:32] := 0; FI;
```

PCMPEQD (With 128-bit Operands)

```
DEST[127:0] := COMPARE_DWORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPEQD (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] := 0
```

VPCMPEQD (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_DWORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPEQD (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
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+31:i] = SRC2[31:0];
                ELSE CMP := SRC1[i+31:i] = SRC2[i+31: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 Equivalents

VPCMPEQB __mmask64 _mm512_cmpeq_epi8_mask(__m512i a, __m512i b);
VPCMPEQB __mmask64 _mm512_mask_cmpeq_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPEQB __mmask32 _mm256_cmpeq_epi8_mask(__m256i a, __m256i b);
VPCMPEQB __mmask32 _mm256_mask_cmpeq_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPEQB __mmask16 _mm_cmpeq_epi8_mask(__m128i a, __m128i b);
VPCMPEQB __mmask16 _mm_mask_cmpeq_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPEQW __mmask32 _mm512_cmpeq_epi16_mask(__m512i a, __m512i b);
VPCMPEQW __mmask32 _mm512_mask_cmpeq_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPEQW __mmask16 _mm256_cmpeq_epi16_mask(__m256i a, __m256i b);
VPCMPEQW __mmask16 _mm256_mask_cmpeq_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPEQW __mmask8 _mm_cmpeq_epi16_mask(__m128i a, __m128i b);
VPCMPEQW __mmask8 _mm_mask_cmpeq_epi16_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPEQD __mmask16 _mm512_cmpeq_epi32_mask(__m512i a, __m512i b);
VPCMPEQD __mmask16 _mm512_mask_cmpeq_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPEQD __mmask8 _mm256_cmpeq_epi32_mask(__m256i a, __m256i b);
VPCMPEQD __mmask8 _mm256_mask_cmpeq_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPEQD __mmask8 _mm_cmpeq_epi32_mask(__m128i a, __m128i b);
VPCMPEQD __mmask8 _mm_mask_cmpeq_epi32_mask(__mmask8 k, __m128i a, __m128i b);
PCMPEQB __m64 _mm_cmpeq_pi8 (__m64 m1, __m64 m2)
PCMPEQW __m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)
PCMPEQD __m64 _mm_cmpeq_pi32 (__m64 m1, __m64 m2)
(V)PCMPEQB __m128i _mm_cmpeq_epi8 (__m128i a, __m128i b)
(V)PCMPEQW __m128i _mm_cmpeq_epi16 (__m128i a, __m128i b)
(V)PCMPEQD __m128i _mm_cmpeq_epi32 (__m128i a, __m128i b)
VPCMPEQB __m256i _mm256_cmpeq_epi8 (__m256i a, __m256i b)
VPCMPEQW __m256i _mm256_cmpeq_epi16 (__m256i a, __m256i b)
VPCMPEQD __m256i _mm256_cmpeq_epi32 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPCMPEQD, see Table 2-49, “Type E4 Class Exception Conditions.”
EVEX-encoded VPCMPEQB/W, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

PCMPEQQ—Compare Packed Qword Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 29 /r PCMPEQQ xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed qwords in xmm2/m128 and xmm1 for equality.
VEX.128.66.0F38.WIG 29 /r VPCMPEQQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed quadwords in xmm3/m128 and xmm2 for equality.
VEX.256.66.0F38.WIG 29 /r VPCMPEQQ ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed quadwords in ymm3/m256 and ymm2 for equality.
EVEX.128.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare Equal between int64 vector xmm2 and int64 vector xmm3/m128/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare Equal between int64 vector ymm2 and int64 vector ymm3/m256/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1[1]	Compare Equal between int64 vector zmm2 and int64 vector zmm3/m512/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs an SIMD compare for equality of the packed quadwords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE 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 destination register remain 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.

EVEX encoded VPCMPEQQ: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPEQQ (With 128-bit Operands)

```
IF (DEST[63:0] = SRC[63:0])
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0; FI;
IF (DEST[127:64] = SRC[127:64])
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0; FI;
DEST[MAXVL-1:128] (Unmodified)
```

COMPARE_QWORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[63:0] = SRC2[63:0]
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0; FI;
    IF SRC1[127:64] = SRC2[127:64]
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0; FI;
```

VPCMPEQQ (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_QWORDS_EQUAL(SRC1,SRC2)
DEST[MAXVL-1:128] := 0
```

VPCMPEQQ (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_QWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_QWORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPEQQ (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] = SRC2[63:0];
                ELSE CMP := SRC1[i+63:i] = 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

VPCMPEQQ __mmask8 _mm512_cmpeq_epi64_mask(__m512i a, __m512i b);
VPCMPEQQ __mmask8 _mm512_mask_cmpeq_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPEQQ __mmask8 _mm256_cmpeq_epi64_mask(__m256i a, __m256i b);
VPCMPEQQ __mmask8 _mm256_mask_cmpeq_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPEQQ __mmask8 _mm_cmpeq_epi64_mask(__m128i a, __m128i b);
VPCMPEQQ __mmask8 _mm_mask_cmpeq_epi64_mask(__mmask8 k, __m128i a, __m128i b);
(V)PCMPEQQ __m128i _mm_cmpeq_epi64(__m128i a, __m128i b);
VPCMPEQQ __m256i _mm256_cmpeq_epi64(__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 64 /r[1] PCMPGTB mm, mm/m64	A	V/V	MMX	Compare packed signed byte integers in mm and mm/m64 for greater than.
66 0F 64 /r PCMPGTB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed byte integers in xmm1 and xmm2/m128 for greater than.
NP 0F 65 /r[1] PCMPGTW mm, mm/m64	A	V/V	MMX	Compare packed signed word integers in mm and mm/m64 for greater than.
66 0F 65 /r PCMPGTW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm1 and xmm2/m128 for greater than.
NP 0F 66 /r[1] PCMPGTD mm, mm/m64	A	V/V	MMX	Compare packed signed doubleword integers in mm and mm/m64 for greater than.
66 0F 66 /r PCMPGTD xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed doubleword integers in xmm1 and xmm2/m128 for greater than.
VEX.128.66.0F.WIG 64 /r VPCMPGTB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than.
VEX.128.66.0F.WIG 65 /r VPCMPGTW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm2 and xmm3/m128 for greater than.
VEX.128.66.0F.WIG 66 /r VPCMPGTD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed doubleword integers in xmm2 and xmm3/m128 for greater than.
VEX.256.66.0F.WIG 64 /r VPCMPGTB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 for greater than.
VEX.256.66.0F.WIG 65 /r VPCMPGTW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm2 and ymm3/m256 for greater than.
VEX.256.66.0F.WIG 66 /r VPCMPGTD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed doubleword integers in ymm2 and ymm3/m256 for greater than.
EVEX.128.66.0F.W0 66 /r VPCMPGTD k1 {k2}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Compare Greater between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.W0 66 /r VPCMPGTD k1 {k2}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Compare Greater between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.W0 66 /r VPCMPGTD k1 {k2}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1[2]	Compare Greater between int32 elements in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask. k2.
EVEX.128.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, xmm2, xmm3/m128	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, ymm2, ymm3/m256	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed signed byte integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, zmm2, zmm3/m512	D	V/V	AVX512BW OR AVX10.1[1]	Compare packed signed byte integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.128.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, xmm2, xmm3/m128	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed signed word integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, ymm2, ymm3/m256	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed signed word integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, zmm2, zmm3/m512	D	V/V	AVX512BW OR AVX10.1[2]	Compare packed signed word integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an SIMD signed compare for the greater value of the packed byte, word, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding date element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The PCMPGTB instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPGTW instruction compares the corresponding signed word integers in the destination and source operands; and the PCMPGTD instruction compares the corresponding signed doubleword integers in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand 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.

EVEX encoded VPCMPGTD: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPGTB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPGTB (With 64-bit Operands)

```
    IF DEST[7:0] > SRC[7:0]
        THEN DEST[7:0) := FFH;
        ELSE DEST[7:0] := 0; FI;
    (* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
    IF DEST[63:56] > SRC[63:56]
        THEN DEST[63:56] := FFH;
        ELSE DEST[63:56] := 0; FI;
```

COMPARE_BYTES_GREATER (SRC1, SRC2)

```
    IF SRC1[7:0] > SRC2[7:0]
    THEN DEST[7:0] := FFH;
    ELSE DEST[7:0] := 0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
    IF SRC1[127:120] > SRC2[127:120]
    THEN DEST[127:120] := FFH;
    ELSE DEST[127:120] := 0; FI;
```

COMPARE_WORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[15:0] > SRC2[15:0]
    THEN DEST[15:0] := FFFFH;
    ELSE DEST[15:0] := 0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
    IF SRC1[127:112] > SRC2[127:112]
    THEN DEST[127:112] := FFFFH;
    ELSE DEST[127:112] := 0; FI;
```

COMPARE_DWORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[31:0] > SRC2[31:0]
    THEN DEST[31:0] := FFFFFFFFH;
    ELSE DEST[31:0] := 0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
    IF SRC1[127:96] > SRC2[127:96]
    THEN DEST[127:96] := FFFFFFFFH;
    ELSE DEST[127:96] := 0; FI;
```

PCMPGTB (With 128-bit Operands)

```
DEST[127:0] := COMPARE_BYTES_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTB (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_BYTES_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] := 0
```

VPCMPGTB (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_BYTES_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_BYTES_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPGTB (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
            /* signed comparison */
            CMP := SRC1[i+7:i] > 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
```

PCMPGTW (With 64-bit Operands)

```
    IF DEST[15:0] > SRC[15:0]
        THEN DEST[15:0] := FFFFH;
        ELSE DEST[15:0] := 0; FI;
    (* Continue comparison of 2nd and 3rd words in DEST and SRC *)
    IF DEST[63:48] > SRC[63:48]
        THEN DEST[63:48] := FFFFH;
        ELSE DEST[63:48] := 0; FI;
```

PCMPGTW (With 128-bit Operands)

```
DEST[127:0] := COMPARE_WORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTW (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_WORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] := 0
```

VPCMPGTW (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_WORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_WORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPGTW (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
            /* signed comparison */
            CMP := SRC1[i+15:i] > SRC2[i+15: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
```

PCMPGTD (With 64-bit Operands)

```
    IF DEST[31:0] > SRC[31:0]
        THEN DEST[31:0] := FFFFFFFFH;
        ELSE DEST[31:0] := 0; FI;
    IF DEST[63:32] > SRC[63:32]
        THEN DEST[63:32] := FFFFFFFFH;
        ELSE DEST[63:32] := 0; FI;
```

PCMPGTD (With 128-bit Operands)

```
DEST[127:0] := COMPARE_DWORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTD (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_DWORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] := 0
```

VPCMPGTD (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_DWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_DWORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPGTD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+31:i] > SRC2[31:0];
                ELSE CMP := SRC1[i+31:i] > SRC2[i+31: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 Equivalents

```
VPCMPGTB __mmask64 _mm512_cmpgt_epi8_mask(__m512i a, __m512i b);
VPCMPGTB __mmask64 _mm512_mask_cmpgt_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPGTB __mmask32 _mm256_cmpgt_epi8_mask(__m256i a, __m256i b);
VPCMPGTB __mmask32 _mm256_mask_cmpgt_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPGTB __mmask16 _mm_cmpgt_epi8_mask(__m128i a, __m128i b);
VPCMPGTB __mmask16 _mm_mask_cmpgt_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPGTD __mmask16 _mm512_cmpgt_epi32_mask(__m512i a, __m512i b);
VPCMPGTD __mmask16 _mm512_mask_cmpgt_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPGTD __mmask8 _mm256_cmpgt_epi32_mask(__m256i a, __m256i b);
VPCMPGTD __mmask8 _mm256_mask_cmpgt_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTD __mmask8 _mm_cmpgt_epi32_mask(__m128i a, __m128i b);
VPCMPGTD __mmask8 _mm_mask_cmpgt_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPGTW __mmask32 _mm512_cmpgt_epi16_mask(__m512i a, __m512i b);
VPCMPGTW __mmask32 _mm512_mask_cmpgt_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPGTW __mmask16 _mm256_cmpgt_epi16_mask(__m256i a, __m256i b);
VPCMPGTW __mmask16 _mm256_mask_cmpgt_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPGTW __mmask8 _mm_cmpgt_epi16_mask(__m128i a, __m128i b);
VPCMPGTW __mmask8 _mm_mask_cmpgt_epi16_mask(__mmask8 k, __m128i a, __m128i b);
PCMPGTB __m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)
PCMPGTW __m64 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)
PCMPGTD __m64 _mm_cmpgt_pi32 (__m64 m1, __m64 m2)
(V)PCMPGTB __m128i _mm_cmpgt_epi8 ( __m128i a, __m128i b)
(V)PCMPGTW __m128i _mm_cmpgt_epi16 ( __m128i a, __m128i b)
(V)DCMPGTD __m128i _mm_cmpgt_epi32 ( __m128i a, __m128i b)
VPCMPGTB __m256i _mm256_cmpgt_epi8 ( __m256i a, __m256i b)
VPCMPGTW __m256i _mm256_cmpgt_epi16 ( __m256i a, __m256i b)
VPCMPGTD __m256i _mm256_cmpgt_epi32 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

EVEX-encoded VPCMPGTB/W, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

PCMPGTQ—Compare Packed Data for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 37 /r PCMPGTQ xmm1,xmm2/m128	A	V/V	SSE4_2	Compare packed signed qwords in xmm2/m128 and xmm1 for greater than.
VEX.128.66.0F38.WIG 37 /r VPCMPGTQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed qwords in xmm2 and xmm3/m128 for greater than.
VEX.256.66.0F38.WIG 37 /r VPCMPGTQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed qwords in ymm2 and ymm3/m256 for greater than.
EVEX.128.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare Greater between int64 vector xmm2 and int64 vector xmm3/m128/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare Greater between int64 vector ymm2 and int64 vector ymm3/m256/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1[1]	Compare Greater between int64 vector zmm2 and int64 vector zmm3/m512/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs an SIMD signed compare for the packed quadwords in the destination operand (first operand) and the source operand (second operand). If the data element in the first (destination) operand is greater than the corresponding element in the second (source) operand, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand 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.

EVEX encoded VPCMPGTD/Q: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

COMPARE_QWORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[63:0] > SRC2[63:0]
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0; FI;
    IF SRC1[127:64] > SRC2[127:64]
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0; FI;
```

VPCMPGTQ (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_QWORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] := 0
```

VPCMPGTQ (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_QWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_QWORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPGTQ (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
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+63:i] > SRC2[63:0];
                ELSE CMP := SRC1[i+63:i] > 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

VPCMPGTQ __mmask8 _mm512_cmpgt_epi64_mask(__m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm512_mask_cmpgt_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm256_cmpgt_epi64_mask(__m256i a, __m256i b);
VPCMPGTQ __mmask8 _mm256_mask_cmpgt_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTQ __mmask8 _mm_cmpgt_epi64_mask(__m128i a, __m128i b);
VPCMPGTQ __mmask8 _mm_mask_cmpgt_epi64_mask(__mmask8 k, __m128i a, __m128i b);
(V)PCMPGTQ __m128i _mm_cmpgt_epi64(__m128i a, __m128i b)
VPCMPGTQ __m256i _mm256_cmpgt_epi64(__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PEXTRB/PEXTRD/PEXTRQ—Extract Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 14 /r ib PEXTRB reg/m8, xmm2, imm8	A	V/V	SSE4_1	Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r32 or r64 are zeroed.
66 0F 3A 16 /r ib PEXTRD r/m32, xmm2, imm8	A	V/V	SSE4_1	Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r/m32.
66 REX.W 0F 3A 16 /r ib PEXTRQ r/m64, xmm2, imm8	A	V/N.E.	SSE4_1	Extract a qword integer value from xmm2 at the source qword offset specified by imm8 into r/m64.
VEX.128.66.0F3A.W0 14 /r ib VPEXTRB reg/m8, xmm2, imm8	A	V[1]/V	AVX	Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 16 /r ib VPEXTRD r32/m32, xmm2, imm8	A	V/V	AVX	Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.
VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ r64/m64, xmm2, imm8	A	V/I[2]	AVX	Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.
EVEX.128.66.0F3A.WIG 14 /r ib VPEXTRB reg/m8, xmm2, imm8	B	V/V	AVX512BW OR AVX10.1[3]	Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F3A.W0 16 /r ib VPEXTRD r32/m32, xmm2, imm8	B	V/V	AVX512DQ OR AVX10.1[3]	Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.
EVEX.128.66.0F3A.W1 16 /r ib VPEXTRQ r64/m64, xmm2, imm8	B	V/N.E.[2]	AVX512DQ OR AVX10.1[3]	Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPEXTRB (similar to legacy REX.W=1 prefix in PEXTRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.
3. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
B	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

Extract a byte/dword/qword integer value from the source XMM register at a byte/dword/qword offset determined from imm8[3:0]. The destination can be a register or byte/dword/qword memory location. If the destination is a register, the upper bits of the register are zero extended.

In legacy non-VEX encoded version and if the destination operand is a register, the default operand size in 64-bit mode for PEXTRB/PEXTRD is 64 bits, the bits above the least significant byte/dword data are filled with zeros. PEXTRQ is not encodable in non-64-bit modes and requires REX.W in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L”L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRB/VPEXTRD is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

Operation

```
CASE of
    PEXTRB: SEL := COUNT[3:0];
            TEMP := (Src >> SEL*8) AND FFH;
            IF (DEST = Mem8)
                THEN
                Mem8 := TEMP[7:0];
            ELSE IF (64-Bit Mode and 64-bit register selected)
                THEN
                    R64[7:0] := TEMP[7:0];
                    r64[63:8] := ZERO_FILL; };
            ELSE
                    R32[7:0] := TEMP[7:0];
                    r32[31:8] := ZERO_FILL; };
            FI;
    PEXTRD:SEL := COUNT[1:0];
            TEMP := (Src >> SEL*32) AND FFFF_FFFFH;
            DEST := TEMP;
    PEXTRQ: SEL := COUNT[0];
            TEMP := (Src >> SEL*64);
            DEST := TEMP;
EASC:
```

VPEXTRTD/VPEXTRQ

```
IF (64-Bit Mode and 64-bit dest operand)
THEN
    Src_Offset := imm8[0]
    r64/m64 := (Src >> Src_Offset * 64)
ELSE
    Src_Offset := imm8[1:0]
    r32/m32 := ((Src >> Src_Offset *32) AND 0FFFFFFFFh);
FI
```

VPEXTRB (dest=m8)

```
SRC_Offset := imm8[3:0]
Mem8 := (Src >> Src_Offset*8)
```

VPEXTRB (dest=reg)

```
IF (64-Bit Mode )
THEN
    SRC_Offset := imm8[3:0]
    DEST[7:0] := ((Src >> Src_Offset*8) AND 0FFh)
    DEST[63:8] := ZERO_FILL;
ELSE
    SRC_Offset := imm8[3:0];
    DEST[7:0] := ((Src >> Src_Offset*8) AND 0FFh);
    DEST[31:8] := ZERO_FILL;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PEXTRB int _mm_extract_epi8 (__m128i src, const int ndx);
PEXTRD int _mm_extract_epi32 (__m128i src, const int ndx);
PEXTRQ __int64 _mm_extract_epi64 (__m128i src, const int ndx);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 1 or EVEX.L’L > 0. If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

PEXTRW—Extract Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C5 /r ib[1] PEXTRW reg, mm, imm8	A	V/V	SSE	Extract the word specified by imm8 from mm and move it to reg, bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F C5 /r ib PEXTRW reg, xmm, imm8	A	V/V	SSE2	Extract the word specified by imm8 from xmm and move it to reg, bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F 3A 15 /r ib PEXTRW reg/m16, xmm, imm8	B	V/V	SSE4_1	Extract the word specified by imm8 from xmm and copy it to lowest 16 bits of reg or m16. Zero-extend the result in the destination, r32 or r64.
VEX.128.66.0F.W0 C5 /r ib VPEXTRW reg, xmm1, imm8	A	V[2]/V	AVX	Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 15 /r ib VPEXTRW reg/m16, xmm2, imm8	B	V/V	AVX	Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F.WIG C5 /r ib VPEXTRW reg, xmm1, imm8	A	V/V	AVX512BW OR AVX10.1[3]	Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F3A.WIG 15 /r ib VPEXTRW reg/m16, xmm2, imm8	C	V/V	AVX512BW OR AVX10.1[3]	Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. In 64-bit mode, VEX.W1 is ignored for VPEXTRW (similar to legacy REX.W=1 prefix in PEXTRW).
3. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
C	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

Copies the word in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand can be an MMX technology register or an XMM register. The destination operand can be the low word of a general-purpose register or a 16-bit memory address. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location. The content of the destination register above bit 16 is cleared (set to all 0s).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). If the destination operand is a general-purpose register, the default operand size is 64-bits in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRW is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

Operation

```
IF (DEST = Mem16)
THEN
    SEL := COUNT[2:0];
    TEMP := (Src >> SEL*16) AND FFFFH;
    Mem16 := TEMP[15:0];
ELSE IF (64-Bit Mode and destination is a general-purpose register)
    THEN
         FOR (PEXTRW instruction with 64-bit source operand)
            {  SEL := COUNT[1:0];
               TEMP := (SRC >> (SEL ∗ 16)) AND FFFFH;
               r64[15:0] := TEMP[15:0];
               r64[63:16] := ZERO_FILL; };
         FOR (PEXTRW instruction with 128-bit source operand)
            {  SEL := COUNT[2:0];
               TEMP := (SRC >> (SEL ∗ 16)) AND FFFFH;
               r64[15:0] := TEMP[15:0];
               r64[63:16] := ZERO_FILL; }
    ELSE
         FOR (PEXTRW instruction with 64-bit source operand)
           {   SEL := COUNT[1:0];
               TEMP := (SRC >> (SEL ∗ 16)) AND FFFFH;
               r32[15:0] := TEMP[15:0];
               r32[31:16] := ZERO_FILL; };
         FOR (PEXTRW instruction with 128-bit source operand)
           {   SEL := COUNT[2:0];
               TEMP := (SRC >> (SEL ∗ 16)) AND FFFFH;
               r32[15:0] := TEMP[15:0];
               r32[31:16] := ZERO_FILL; };
    FI;
FI;
```

VPEXTRW (dest=m16)

```
SRC_Offset := imm8[2:0]
Mem16 := (Src >> Src_Offset*16)
```

VPEXTRW (dest=reg)

```
IF (64-Bit Mode )
THEN
    SRC_Offset := imm8[2:0]
    DEST[15:0] := ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[63:16] := ZERO_FILL;
ELSE
    SRC_Offset := imm8[2:0]
    DEST[15:0] := ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[31:16] := ZERO_FILL;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PEXTRW int _mm_extract_pi16 (__m64 a, int n)
PEXTRW int _mm_extract_epi16 (__m128i a, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 1 or EVEX.L’L > 0.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

PINSRB/PINSRD/PINSRQ—Insert Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 20 /r ib PINSRB xmm1, r32/m8, imm8	A	V/V	SSE4_1	Insert a byte integer value from r32/m8 into xmm1 at the destination element in xmm1 specified by imm8.
66 0F 3A 22 /r ib PINSRD xmm1, r/m32, imm8	A	V/V	SSE4_1	Insert a dword integer value from r/m32 into the xmm1 at the destination element specified by imm8.
66 REX.W 0F 3A 22 /r ib PINSRQ xmm1, r/m64, imm8	A	V/N. E.	SSE4_1	Insert a qword integer value from r/m64 into the xmm1 at the destination element specified by imm8.
VEX.128.66.0F3A.W0 20 /r ib VPINSRB xmm1, xmm2, r32/m8, imm8	B	V[1]/V	AVX	Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.
VEX.128.66.0F3A.W0 22 /r ib VPINSRD xmm1, xmm2, r/m32, imm8	B	V/V	AVX	Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8.
VEX.128.66.0F3A.W1 22 /r ib VPINSRQ xmm1, xmm2, r/m64, imm8	B	V/I[2]	AVX	Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8.
EVEX.128.66.0F3A.WIG 20 /r ib VPINSRB xmm1, xmm2, r32/m8, imm8	C	V/V	AVX512BW OR AVX10.1[3]	Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.
EVEX.128.66.0F3A.W0 22 /r ib VPINSRD xmm1, xmm2, r32/m32, imm8	C	V/V	AVX512DQ OR AVX10.1[3]	Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8.
EVEX.128.66.0F3A.W1 22 /r ib VPINSRQ xmm1, xmm2, r64/m64, imm8	C	V/N.E.[2]	AVX512DQ OR AVX10.1[3]	Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8.

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPINSRB (similar to legacy REX.W=1 prefix with PINSRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.
3. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Copies a byte/dword/qword from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other elements in the destination register are left untouched.) The source operand can be a general-purpose register or a memory location. (When the source operand is a general-purpose register, PINSRB copies the low byte of the register.) The destination operand is an XMM register. The count operand is an 8-bit immediate. When specifying a qword[dword, byte] location in an XMM register, the [2, 4] least-significant bit(s) of the count operand specify the location.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64 bit general purpose registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. VEX.L must be 0, otherwise the instruction will #UD. Attempt to execute VPINSRQ in non-64-bit mode will cause #UD.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. EVEX.L'L must be 0, otherwise the instruction will #UD.

Operation

```
CASE OF
    PINSRB:  SEL := COUNT[3:0];
             MASK := (0FFH << (SEL * 8));
             TEMP := (((SRC[7:0] << (SEL *8)) AND MASK);
    PINSRD:  SEL := COUNT[1:0];
             MASK := (0FFFFFFFFH << (SEL * 32));
             TEMP := (((SRC << (SEL *32)) AND MASK)   ;
    PINSRQ:  SEL := COUNT[0]
             MASK := (0FFFFFFFFFFFFFFFFH << (SEL * 64));
             TEMP := (((SRC << (SEL *64)) AND MASK)   ;
ESAC;
         DEST := ((DEST AND NOT MASK) OR TEMP);
```

VPINSRB (VEX/EVEX Encoded Version)

```
SEL := imm8[3:0]
DEST[127:0] := write_b_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] := 0
```

VPINSRD (VEX/EVEX Encoded Version)

```
SEL := imm8[1:0]
DEST[127:0] := write_d_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] := 0
```

VPINSRQ (VEX/EVEX Encoded Version)

```
SEL := imm8[0]
DEST[127:0] := write_q_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
PINSRB __m128i _mm_insert_epi8 (__m128i s1, int s2, const int ndx);
PINSRD __m128i _mm_insert_epi32 (__m128i s2, int s, const int ndx);
PINSRQ __m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 1 or EVEX.L’L > 0.

PINSRW—Insert Word

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F C4 /r ib[1] PINSRW mm, r32/m16, imm8	A	V/V	SSE	Insert the low word from r32 or from m16 into mm at the word position specified by imm8.
66 0F C4 /r ib PINSRW xmm, r32/m16, imm8	A	V/V	SSE2	Move the low word of r32 or from m16 into xmm at the word position specified by imm8.
VEX.128.66.0F.W0 C4 /r ib VPINSRW xmm1, xmm2, r32/m16, imm8	B	V[2]/V	AVX	Insert the word from r32/m16 at the offset indicated by imm8 into the value from xmm2 and store result in xmm1.
EVEX.128.66.0F.WIG C4 /r ib VPINSRW xmm1, xmm2, r32/m16, imm8	C	V/V	AVX512BW OR AVX10.1[3]	Insert the word from r32/m16 at the offset indicated by imm8 into the value from xmm2 and store result in xmm1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. In 64-bit mode, VEX.W1 is ignored for VPINSRW (similar to legacy REX.W=1 prefix in PINSRW).
3. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Three operand MMX and SSE instructions:

Copies a word from the source operand and inserts it in the destination operand at the location specified with the count operand. (The other words in the destination register are left untouched.) The source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The destination operand can be an MMX technology register or an XMM register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location.

Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

Four operand AVX and AVX-512 instructions:

Combines a word from the first source operand with the second source operand, and inserts it in the destination operand at the location specified with the count operand. The second source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The first source and destination operands are XMM registers. The count operand is an 8-bit immediate. When specifying a word location, the 3 least-significant bits specify the location.

Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L/EVEX.L'L must be 0, otherwise the instruction will #UD.

Operation

PINSRW dest, src, imm8 (MMX)

```
    SEL := imm8[1:0]
    DEST.word[SEL] := src.word[0]
```

PINSRW dest, src, imm8 (SSE)

```
    SEL := imm8[2:0]
    DEST.word[SEL] := src.word[0]
```

VPINSRW dest, src1, src2, imm8 (AVX/AVX512)

```
    SEL := imm8[2:0]
    DEST := src1
    DEST.word[SEL] := src2.word[0]
    DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PINSRW __m64 _mm_insert_pi16 (__m64 a, int d, int n)
PINSRW __m128i _mm_insert_epi16 (__m128i a, int b, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 1 or EVEX.L’L > 0.

PMADDUBSW—Multiply and Add Packed Signed and Unsigned Bytes

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 38 04 /r[1] PMADDUBSW mm1, mm2/m64	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to mm1.
66 0F 38 04 /r PMADDUBSW xmm1, xmm2/m128	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1.
VEX.128.66.0F38.WIG 04 /r VPMADDUBSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1.
VEX.256.66.0F38.WIG 04 /r VPMADDUBSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1.
EVEX.128.66.0F38.WIG 04 /r VPMADDUBSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 04 /r VPMADDUBSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 04 /r VPMADDUBSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

(V)PMADDUBSW multiplies vertically each unsigned byte of the destination operand (first operand) with the corresponding signed byte of the source operand (second operand), producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7-0) in the source and destination operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15-8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15-0). The same operation is performed on the other pairs of adjacent bytes. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

Operation

PMADDUBSW (With 64-bit Operands)

```
    DEST[15-0] = SaturateToSignedWord(SRC[15-8]*DEST[15-8]+SRC[7-0]*DEST[7-0]);
    DEST[31-16] = SaturateToSignedWord(SRC[31-24]*DEST[31-24]+SRC[23-16]*DEST[23-16]);
    DEST[47-32] = SaturateToSignedWord(SRC[47-40]*DEST[47-40]+SRC[39-32]*DEST[39-32]);
    DEST[63-48] = SaturateToSignedWord(SRC[63-56]*DEST[63-56]+SRC[55-48]*DEST[55-48]);
```

PMADDUBSW (With 128-bit Operands)

```
    DEST[15-0] = SaturateToSignedWord(SRC[15-8]* DEST[15-8]+SRC[7-0]*DEST[7-0]);
    // Repeat operation for 2nd through 7th word
    SRC1/DEST[127-112] = SaturateToSignedWord(SRC[127-120]*DEST[127-120]+ SRC[119-112]* DEST[119-112]);
```

VPMADDUBSW (VEX.128 Encoded Version)

```
DEST[15:0] := SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 7th word
DEST[127:112] := SaturateToSignedWord(SRC2[127:120]*SRC1[127:120]+ SRC2[119:112]* SRC1[119:112])
DEST[MAXVL-1:128] := 0
```

VPMADDUBSW (VEX.256 Encoded Version)

```
DEST[15:0] := SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 15th word
DEST[255:240] := SaturateToSignedWord(SRC2[255:248]*SRC1[255:248]+ SRC2[247:240]* SRC1[247:240])
DEST[MAXVL-1:256] := 0
```

VPMADDUBSW (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] := SaturateToSignedWord(SRC2[i+15:i+8]* SRC1[i+15:i+8] + SRC2[i+7:i]*SRC1[i+7:i])
        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] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMADDUBSW __m512i _mm512_maddubs_epi16(__m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_mask_maddubs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_maskz_maddubs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m256i _mm256_mask_maddubs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m256i _mm256_maskz_maddubs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m128i _mm_mask_maddubs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDUBSW __m128i _mm_maskz_maddubs_epi16(__mmask8 k, __m128i a, __m128i b);
PMADDUBSW __m64 _mm_maddubs_pi16 (__m64 a, __m64 b)
(V)PMADDUBSW __m128i _mm_maddubs_epi16 (__m128i a, __m128i b)
VPMADDUBSW __m256i _mm256_maddubs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

PMADDWD—Multiply and Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F5 /r[1] PMADDWD mm, mm/m64	A	V/V	MMX	Multiply the packed words in mm by the packed words in mm/m64, add adjacent doubleword results, and store in mm.
66 0F F5 /r PMADDWD xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed word integers in xmm1 by the packed word integers in xmm2/m128, add adjacent doubleword results, and store in xmm1.
VEX.128.66.0F.WIG F5 /r VPMADDWD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1.
VEX.256.66.0F.WIG F5 /r VPMADDWD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed word integers in ymm2 by the packed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1.
EVEX.128.66.0F.WIG F5 /r VPMADDWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1 under writemask k1.
EVEX.256.66.0F.WIG F5 /r VPMADDWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Multiply the packed word integers in ymm2 by the packed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1 under writemask k1.
EVEX.512.66.0F.WIG F5 /r VPMADDWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Multiply the packed word integers in zmm2 by the packed word integers in zmm3/m512, add adjacent doubleword results, and store in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 1-11 shows this operation when using 64-bit operands).

The (V)PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The first source and destination operands are MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

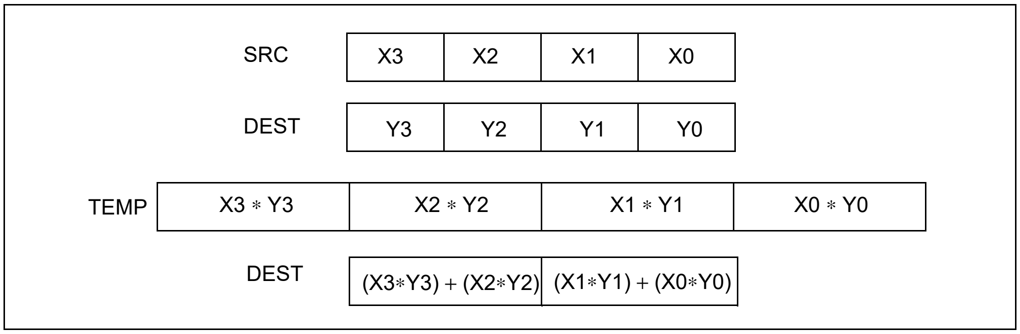


Figure 1-11. PMADDWD Execution Model Using 64-bit Operands

Operation

PMADDWD (With 64-bit Operands)

```
    DEST[31:0] := (DEST[15:0] ∗ SRC[15:0]) + (DEST[31:16] ∗ SRC[31:16]);
    DEST[63:32] := (DEST[47:32] ∗ SRC[47:32]) + (DEST[63:48] ∗ SRC[63:48]);
```

PMADDWD (With 128-bit Operands)

```
    DEST[31:0] := (DEST[15:0] ∗ SRC[15:0]) + (DEST[31:16] ∗ SRC[31:16]);
    DEST[63:32] := (DEST[47:32] ∗ SRC[47:32]) + (DEST[63:48] ∗ SRC[63:48]);
    DEST[95:64] := (DEST[79:64] ∗ SRC[79:64]) + (DEST[95:80] ∗ SRC[95:80]);
    DEST[127:96] := (DEST[111:96] ∗ SRC[111:96]) + (DEST[127:112] ∗ SRC[127:112]);
```

VPMADDWD (VEX.128 Encoded Version)

```
DEST[31:0] := (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] := (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] := (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] := (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[MAXVL-1:128] := 0
```

```
VPMADDWD (VEX.256 Encoded Version)
DEST[31:0] := (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] := (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] := (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] := (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[159:128] := (SRC1[143:128] * SRC2[143:128]) + (SRC1[159:144] * SRC2[159:144])
DEST[191:160] := (SRC1[175:160] * SRC2[175:160]) + (SRC1[191:176] * SRC2[191:176])
DEST[223:192] := (SRC1[207:192] * SRC2[207:192]) + (SRC1[223:208] * SRC2[223:208])
DEST[255:224] := (SRC1[239:224] * SRC2[239:224]) + (SRC1[255:240] * SRC2[255:240])
DEST[MAXVL-1:256] := 0

VPMADDWD (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] := (SRC2[i+31:i+16]* SRC1[i+31:i+16]) + (SRC2[i+15:i]*SRC1[i+15: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:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMADDWD __m512i _mm512_madd_epi16( __m512i a, __m512i b);
VPMADDWD __m512i _mm512_mask_madd_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMADDWD __m512i _mm512_maskz_madd_epi16( __mmask32 k, __m512i a, __m512i b);
VPMADDWD __m256i _mm256_mask_madd_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDWD __m256i _mm256_maskz_madd_epi16( __mmask16 k, __m256i a, __m256i b);
VPMADDWD __m128i _mm_mask_madd_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDWD __m128i _mm_maskz_madd_epi16( __mmask8 k, __m128i a, __m128i b);
PMADDWD __m64 _mm_madd_pi16(__m64 m1, __m64 m2)
(V)PMADDWD __m128i _mm_madd_epi16 ( __m128i a, __m128i b)
VPMADDWD __m256i _mm256_madd_epi16 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

PMAXSB/PMAXSW/PMAXSD/PMAXSQ—Maximum of Packed Signed Integers

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F EE /r[1] PMAXSW mm1, mm2/m64	A	V/V	SSE	Compare signed word integers in mm2/m64 and mm1 and return maximum values.
66 0F 38 3C /r PMAXSB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F EE /r PMAXSW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
66 0F 38 3D /r PMAXSD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.128.66.0F38.WIG 3C /r VPMAXSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.128.66.0F.WIG EE /r VPMAXSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and store packed maximum values in xmm1.
VEX.128.66.0F38.WIG 3D /r VPMAXSD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.256.66.0F38.WIG 3C /r VPMAXSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.256.66.0F.WIG EE /r VPMAXSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and store packed maximum values in ymm1.
VEX.256.66.0F38.WIG 3D /r VPMAXSD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.128.66.0F38.WIG 3C /r VPMAXSB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 3C /r VPMAXSB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 3C /r VPMAXSB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[1]	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.128.66.0F.WIG EE /r VPMAXSW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F.WIG EE /r VPMAXSW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F.WIG EE /r VPMAXSW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[1]	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EVEX.128.66.0F38.W0 3D /r VPMAXSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed signed dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 3D /r VPMAXSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed signed dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 3D /r VPMAXSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F OR AVX10.1[1]	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 3D /r VPMAXSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed signed qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 3D /r VPMAXSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed signed qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 3D /r VPMAXSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F OR AVX10.1[1]	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed signed byte, word, dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is 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 second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPMAXSD/Q: 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 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

EVEX encoded VPMAXSB/W: 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. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXSW (64-bit Operands)

```
    IF DEST[15:0] > SRC[15:0]) THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd and 3rd words in source and destination operands *)
    IF DEST[63:48] > SRC[63:48]) THEN
        DEST[63:48] := DEST[63:48];
    ELSE
        DEST[63:48] := SRC[63:48]; FI;
```

PMAXSB (128-bit Legacy SSE Version)

```
    IF DEST[7:0] > SRC[7:0] THEN
        DEST[7:0] := DEST[7:0];
    ELSE
        DEST[7:0] := SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] >SRC[127:120] THEN
        DEST[127:120] := DEST[127:120];
    ELSE
        DEST[127:120] := SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXSB (VEX.128 Encoded Version)

```
    IF SRC1[7:0] > SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[7:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] >SRC2[127:120] THEN
        DEST[127:120] := SRC1[127:120];
    ELSE
        DEST[127:120] := SRC2[127:120]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXSB (VEX.256 Encoded Version)

```
    IF SRC1[7:0] > SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[7:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] >SRC2[255:248] THEN
        DEST[255:248] := SRC1[255:248];
    ELSE
        DEST[255:248] := SRC2[255:248]; FI;
DEST[MAXVL-1:256] := 0
```

VPMAXSB (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
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] := SRC1[i+7:i];
            ELSE DEST[i+7:i] := SRC2[i+7:i];
        FI;
        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
```

PMAXSW (128-bit Legacy SSE Version)

```
    IF DEST[15:0] >SRC[15:0] THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] >SRC[127:112] THEN
        DEST[127:112] := DEST[127:112];
    ELSE
        DEST[127:112] := SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXSW (VEX.128 Encoded Version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] >SRC2[127:112] THEN
        DEST[127:112] := SRC1[127:112];
    ELSE
        DEST[127:112] := SRC2[127:112]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXSW (VEX.256 Encoded Version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] >SRC2[255:240] THEN
        DEST[255:240] := SRC1[255:240];
    ELSE
        DEST[255:240] := SRC2[255:240]; FI;
DEST[MAXVL-1:256] := 0
```

VPMAXSW (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
        IF SRC1[i+15:i] > SRC2[i+15:i]
            THEN DEST[i+15:i] := SRC1[i+15:i];
            ELSE DEST[i+15:i] := SRC2[i+15:i];
        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
```

PMAXSD (128-bit Legacy SSE Version)

```
    IF DEST[31:0] >SRC[31:0] THEN
        DEST[31:0] := DEST[31:0];
    ELSE
        DEST[31:0] := SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] >SRC[127:96] THEN
        DEST[127:96] := DEST[127:96];
    ELSE
        DEST[127:96] := SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXSD (VEX.128 Encoded Version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] > SRC2[127:96] THEN
        DEST[127:96] := SRC1[127:96];
    ELSE
        DEST[127:96] := SRC2[127:96]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXSD (VEX.256 Encoded Version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] > SRC2[255:224] THEN
        DEST[255:224] := SRC1[255:224];
    ELSE
        DEST[255:224] := SRC2[255:224]; FI;
DEST[MAXVL-1:256] := 0
```

VPMAXSD (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
                IF SRC1[i+31:i] > SRC2[31:0]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[i+31:i];
            FI;
        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
```

VPMAXSQ (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
                IF SRC1[i+63:i] > SRC2[63:0]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] > SRC2[i+63:i]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[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
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMAXSB __m512i _mm512_max_epi8(__m512i a, __m512i b);
VPMAXSB __m512i _mm512_mask_max_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXSB __m512i _mm512_maskz_max_epi8(__mmask64 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_max_epi16(__m512i a, __m512i b);
VPMAXSW __m512i _mm512_mask_max_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_maskz_max_epi16(__mmask32 k, __m512i a, __m512i b);
VPMAXSB __m256i _mm256_mask_max_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXSB __m256i _mm256_maskz_max_epi8(__mmask32 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_mask_max_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_maskz_max_epi16(__mmask16 k, __m256i a, __m256i b);
VPMAXSB __m128i _mm_mask_max_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXSB __m128i _mm_maskz_max_epi8(__mmask16 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_mask_max_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_maskz_max_epi16(__mmask8 k, __m128i a, __m128i b);
VPMAXSD __m256i _mm256_mask_max_epi32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSD __m256i _mm256_maskz_max_epi32(__mmask16 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_mask_max_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_maskz_max_epi64(__mmask8 k, __m256i a, __m256i b);
VPMAXSD __m128i _mm_mask_max_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSD __m128i _mm_maskz_max_epi32(__mmask8 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_mask_max_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_maskz_max_epu64(__mmask8 k, __m128i a, __m128i b);
VPMAXSD __m512i _mm512_max_epi32(__m512i a, __m512i b);
VPMAXSD __m512i _mm512_mask_max_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMAXSD __m512i _mm512_maskz_max_epi32(__mmask16 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_max_epi64(__m512i a, __m512i b);
VPMAXSQ __m512i _mm512_mask_max_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_maskz_max_epi64(__mmask8 k, __m512i a, __m512i b);
(V)PMAXSB __m128i _mm_max_epi8 (__m128i a, __m128i b);
(V)PMAXSW __m128i _mm_max_epi16 (__m128i a, __m128i b)
(V)PMAXSD __m128i _mm_max_epi32 (__m128i a, __m128i b);
VPMAXSB __m256i _mm256_max_epi8 (__m256i a, __m256i b);
VPMAXSW __m256i _mm256_max_epi16 (__m256i a, __m256i b)
VPMAXSD __m256i _mm256_max_epi32 (__m256i a, __m256i b);
PMAXSW:__m64 _mm_max_pi16(__m64 a, __m64 b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPMAXSD/Q, see Table 2-49, “Type E4 Class Exception Conditions.”
EVEX-encoded VPMAXSB/W, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

PMAXUB/PMAXUW—Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DE /r[1] PMAXUB mm1, mm2/m64	A	V/V	SSE	Compare unsigned byte integers in mm2/m64 and mm1 and returns maximum values.
66 0F DE /r PMAXUB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F 38 3E/r PMAXUW xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
VEX.128.66.0F DE /r VPMAXUB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.128.66.0F38 3E/r VPMAXUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and store maximum packed values in xmm1.
VEX.256.66.0F DE /r VPMAXUB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.256.66.0F38 3E/r VPMAXUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and store maximum packed values in ymm1.
EVEX.128.66.0F.WIG DE /r VPMAXUB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F.WIG DE /r VPMAXUB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F.WIG DE /r VPMAXUB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.128.66.0F38.WIG 3E /r VPMAXUW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed unsigned word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 3E /r VPMAXUW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed unsigned word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 3E /r VPMAXUW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Compare packed unsigned word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed unsigned byte, word integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is 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 second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: 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.

Operation

PMAXUB (64-bit Operands)

```
    IF DEST[7:0] > SRC[17:0]) THEN
        DEST[7:0] := DEST[7:0];
    ELSE
        DEST[7:0] := SRC[7:0]; FI;
    (* Repeat operation for 2nd through 7th bytes in source and destination operands *)
    IF DEST[63:56] > SRC[63:56]) THEN
        DEST[63:56] := DEST[63:56];
    ELSE
        DEST[63:56] := SRC[63:56]; FI;
```

PMAXUB (128-bit Legacy SSE Version)

```
    IF DEST[7:0] >SRC[7:0] THEN
        DEST[7:0] := DEST[7:0];
    ELSE
        DEST[15:0] := SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] >SRC[127:120] THEN
        DEST[127:120] := DEST[127:120];
    ELSE
        DEST[127:120] := SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXUB (VEX.128 Encoded Version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[7:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] >SRC2[127:120] THEN
        DEST[127:120] := SRC1[127:120];
    ELSE
        DEST[127:120] := SRC2[127:120]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXUB (VEX.256 Encoded Version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[15:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] >SRC2[255:248] THEN
        DEST[255:248] := SRC1[255:248];
    ELSE
        DEST[255:248] := SRC2[255:248]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXUB (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
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] := SRC1[i+7:i];
            ELSE DEST[i+7:i] := SRC2[i+7:i];
        FI;
        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
```

PMAXUW (128-bit Legacy SSE Version)

```
    IF DEST[15:0] >SRC[15:0] THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] >SRC[127:112] THEN
        DEST[127:112] := DEST[127:112];
    ELSE
        DEST[127:112] := SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXUW (VEX.128 Encoded Version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] >SRC2[127:112] THEN
        DEST[127:112] := SRC1[127:112];
    ELSE
        DEST[127:112] := SRC2[127:112]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXUW (VEX.256 Encoded Version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] >SRC2[255:240] THEN
        DEST[255:240] := SRC1[255:240];
    ELSE
        DEST[255:240] := SRC2[255:240]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXUW (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
        IF SRC1[i+15:i] > SRC2[i+15:i]
            THEN DEST[i+15:i] := SRC1[i+15:i];
            ELSE DEST[i+15:i] := SRC2[i+15:i];
        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
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMAXUB __m512i _mm512_max_epu8(__m512i a, __m512i b);
VPMAXUB __m512i _mm512_mask_max_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXUB __m512i _mm512_maskz_max_epu8(__mmask64 k, __m512i a, __m512i b);
VPMAXUW __m512i _mm512_max_epu16(__m512i a, __m512i b);
VPMAXUW __m512i _mm512_mask_max_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXUW __m512i _mm512_maskz_max_epu16(__mmask32 k, __m512i a, __m512i b);
VPMAXUB __m256i _mm256_mask_max_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXUB __m256i _mm256_maskz_max_epu8(__mmask32 k, __m256i a, __m256i b);
VPMAXUW __m256i _mm256_mask_max_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUW __m256i _mm256_maskz_max_epu16(__mmask16 k, __m256i a, __m256i b);
VPMAXUB __m128i _mm_mask_max_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXUB __m128i _mm_maskz_max_epu8(__mmask16 k, __m128i a, __m128i b);
VPMAXUW __m128i _mm_mask_max_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUW __m128i _mm_maskz_max_epu16(__mmask8 k, __m128i a, __m128i b);
(V)PMAXUB __m128i _mm_max_epu8 (__m128i a, __m128i b);
(V)PMAXUW __m128i _mm_max_epu16 (__m128i a, __m128i b)
VPMAXUB __m256i _mm256_max_epu8 (__m256i a, __m256i b);
VPMAXUW __m256i _mm256_max_epu16 (__m256i a, __m256i b);
PMAXUB __m64 _mm_max_pu8(__m64 a, __m64 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PMAXUD/PMAXUQ—Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3F /r PMAXUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.128.66.0F38.WIG 3F /r VPMAXUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.256.66.0F38.WIG 3F /r VPMAXUD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.128.66.0F38.W0 3F /r VPMAXUD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 3F /r VPMAXUD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 3F /r VPMAXUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1[1]	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 3F /r VPMAXUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 3F /r VPMAXUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 3F /r VPMAXUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1[1]	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed unsigned dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

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

EVEX encoded versions: 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 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXUD (128-bit Legacy SSE Version)

```
    IF DEST[31:0] >SRC[31:0] THEN
        DEST[31:0] := DEST[31:0];
    ELSE
        DEST[31:0] := SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] >SRC[127:96] THEN
        DEST[127:96] := DEST[127:96];
    ELSE
        DEST[127:96] := SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXUD (VEX.128 Encoded Version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] > SRC2[127:96] THEN
        DEST[127:96] := SRC1[127:96];
    ELSE
        DEST[127:96] := SRC2[127:96]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXUD (VEX.256 Encoded Version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] > SRC2[255:224] THEN
        DEST[255:224] := SRC1[255:224];
    ELSE
        DEST[255:224] := SRC2[255:224]; FI;
DEST[MAXVL-1:256] := 0
```

VPMAXUD (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
                IF SRC1[i+31:i] > SRC2[31:0]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[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
                    THEN DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPMAXUQ (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
                IF SRC1[i+63:i] > SRC2[63:0]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[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
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMAXUD __m512i _mm512_max_epu32(__m512i a, __m512i b);
VPMAXUD __m512i _mm512_mask_max_epu32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMAXUD __m512i _mm512_maskz_max_epu32(__mmask16 k, __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_max_epu64(__m512i a, __m512i b);
VPMAXUQ __m512i _mm512_mask_max_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_maskz_max_epu64(__mmask8 k, __m512i a, __m512i b);
VPMAXUD __m256i _mm256_mask_max_epu32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUD __m256i _mm256_maskz_max_epu32(__mmask16 k, __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_mask_max_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_maskz_max_epu64(__mmask8 k, __m256i a, __m256i b);
VPMAXUD __m128i _mm_mask_max_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUD __m128i _mm_maskz_max_epu32(__mmask8 k, __m128i a, __m128i b);
VPMAXUQ __m128i _mm_mask_max_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUQ __m128i _mm_maskz_max_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMAXUD __m128i _mm_max_epu32 (__m128i a, __m128i b);
VPMAXUD __m256i _mm256_max_epu32 (__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-49, "Type E4 Class Exception Conditions."

PMINSB/PMINSW—Minimum of Packed Signed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EA /r[1] PMINSW mm1, mm2/m64	A	V/V	SSE	Compare signed word integers in mm2/m64 and mm1 and return minimum values.
66 0F 38 38 /r PMINSB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F EA /r PMINSW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
VEX.128.66.0F38 38 /r VPMINSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.128.66.0F EA /r VPMINSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.256.66.0F38 38 /r VPMINSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
VEX.256.66.0F EA /r VPMINSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
EVEX.128.66.0F38.WIG 38 /r VPMINSB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 38 /r VPMINSB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 38 /r VPMINSB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F.WIG EA /r VPMINSW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F.WIG EA /r VPMINSW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F.WIG EA /r VPMINSW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed signed byte, word, or dword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is 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 second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: 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.

Operation

PMINSW (64-bit Operands)

```
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd and 3rd words in source and destination operands *)
    IF DEST[63:48] < SRC[63:48] THEN
        DEST[63:48] := DEST[63:48];
    ELSE
        DEST[63:48] := SRC[63:48]; FI;
```

PMINSB (128-bit Legacy SSE Version)

```
    IF DEST[7:0] < SRC[7:0] THEN
        DEST[7:0] := DEST[7:0];
    ELSE
        DEST[15:0] := SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] < SRC[127:120] THEN
        DEST[127:120] := DEST[127:120];
    ELSE
        DEST[127:120] := SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINSB (VEX.128 Encoded Version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
```

```
        DEST[7:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] < SRC2[127:120] THEN
        DEST[127:120] := SRC1[127:120];
    ELSE
        DEST[127:120] := SRC2[127:120]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINSB (VEX.256 Encoded Version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[15:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] < SRC2[255:248] THEN
        DEST[255:248] := SRC1[255:248];
    ELSE
        DEST[255:248] := SRC2[255:248]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINSB (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
        IF SRC1[i+7:i] < SRC2[i+7:i]
            THEN DEST[i+7:i] := SRC1[i+7:i];
            ELSE DEST[i+7:i] := SRC2[i+7:i];
        FI;
        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
```

PMINSW (128-bit Legacy SSE Version)

```
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] < SRC[127:112] THEN
        DEST[127:112] := DEST[127:112];
    ELSE
        DEST[127:112] := SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINSW (VEX.128 Encoded Version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
```

```
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] < SRC2[127:112] THEN
        DEST[127:112] := SRC1[127:112];
    ELSE
        DEST[127:112] := SRC2[127:112]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINSW (VEX.256 Encoded Version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] < SRC2[255:240] THEN
        DEST[255:240] := SRC1[255:240];
    ELSE
        DEST[255:240] := SRC2[255:240]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINSW (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
        IF SRC1[i+15:i] < SRC2[i+15:i]
            THEN DEST[i+15:i] := SRC1[i+15:i];
            ELSE DEST[i+15:i] := SRC2[i+15:i];
        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
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINSB __m512i _mm512_min_epi8(__m512i a, __m512i b);
VPMINSB __m512i _mm512_mask_min_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINSB __m512i _mm512_maskz_min_epi8(__mmask64 k, __m512i a, __m512i b);
VPMINSW __m512i _mm512_min_epi16(__m512i a, __m512i b);
VPMINSW __m512i _mm512_mask_min_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINSW __m512i _mm512_maskz_min_epi16(__mmask32 k, __m512i a, __m512i b);
VPMINSB __m256i _mm256_mask_min_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINSB __m256i _mm256_maskz_min_epi8(__mmask32 k, __m256i a, __m256i b);
VPMINSW __m256i _mm256_mask_min_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINSW __m256i _mm256_maskz_min_epi16(__mmask16 k, __m256i a, __m256i b);
VPMINSB __m128i _mm_mask_min_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMINSB __m128i _mm_maskz_min_epi8(__mmask16 k, __m128i a, __m128i b);

VPMINSW __m128i _mm_mask_min_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSW __m128i _mm_maskz_min_epi16(__mmask8 k, __m128i a, __m128i b);
(V)PMINSB __m128i _mm_min_epi8 (__m128i a, __m128i b);
(V)PMINSW __m128i _mm_min_epi16 (__m128i a, __m128i b)
VPMINSB __m256i _mm256_min_epi8 (__m256i a, __m256i b);
VPMINSW __m256i _mm256_min_epi16 (__m256i a, __m256i b)
PMINSW__m64 _mm_min_pi16 (__m64 a, __m64 b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

Additionally:

#MF	(64-bit operations only) If there is a pending x87 FPU exception.

PMINSD/PMINSQ—Minimum of Packed Signed Integers

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
66 0F 38 39 /r PMINSD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.128.66.0F38.WIG 39 /r VPMINSD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.256.66.0F38.WIG 39 /r VPMINSD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m128 and store packed minimum values in ymm1.
EVEX.128.66.0F38.W0 39 /r VPMINSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 39 /r VPMINSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 39 /r VPMINSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1[1]	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 39 /r VPMINSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed signed qword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 39 /r VPMINSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed signed qword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 39 /r VPMINSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1[1]	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed signed dword or qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is 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 second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: 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 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINSD (128-bit Legacy SSE Version)

```
    IF DEST[31:0] < SRC[31:0] THEN
        DEST[31:0] := DEST[31:0];
    ELSE
        DEST[31:0] := SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] < SRC[127:96] THEN
        DEST[127:96] := DEST[127:96];
    ELSE
        DEST[127:96] := SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINSD (VEX.128 Encoded Version)

```
    IF SRC1[31:0] < SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] < SRC2[127:96] THEN
        DEST[127:96] := SRC1[127:96];
    ELSE
        DEST[127:96] := SRC2[127:96]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINSD (VEX.256 Encoded Version)

```
    IF SRC1[31:0] < SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] < SRC2[255:224] THEN
        DEST[255:224] := SRC1[255:224];
    ELSE
        DEST[255:224] := SRC2[255:224]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINSD (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
                IF SRC1[i+31:i] < SRC2[31:0]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] < SRC2[i+31:i]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[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
```

VPMINSQ (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
                IF SRC1[i+63:i] < SRC2[63:0]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] < SRC2[i+63:i]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[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
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINSD __m512i _mm512_min_epi32(__m512i a, __m512i b);
VPMINSD __m512i _mm512_mask_min_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINSD __m512i _mm512_maskz_min_epi32(__mmask16 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_min_epi64(__m512i a, __m512i b);
VPMINSQ __m512i _mm512_mask_min_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_maskz_min_epi64(__mmask8 k, __m512i a, __m512i b);
VPMINSD __m256i _mm256_mask_min_epi32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINSD __m256i _mm256_maskz_min_epi32(__mmask16 k, __m256i a, __m256i b);
VPMINSQ __m256i _mm256_mask_min_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMINSQ __m256i _mm256_maskz_min_epi64(__mmask8 k, __m256i a, __m256i b);
VPMINSD __m128i _mm_mask_min_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSD __m128i _mm_maskz_min_epi32(__mmask8 k, __m128i a, __m128i b);
VPMINSQ __m128i _mm_mask_min_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSQ __m128i _mm_maskz_min_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMINSD __m128i _mm_min_epi32 (__m128i a, __m128i b);
VPMINSD __m256i _mm256_min_epi32 (__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PMINUB/PMINUW—Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DA /r[1] PMINUB mm1, mm2/m64	A	V/V	SSE	Compare unsigned byte integers in mm2/m64 and mm1 and returns minimum values.
66 0F DA /r PMINUB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F 38 3A/r PMINUW xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
VEX.128.66.0F DA /r VPMINUB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.128.66.0F38 3A/r VPMINUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.256.66.0F DA /r VPMINUB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
VEX.256.66.0F38 3A/r VPMINUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
EVEX.128.66.0F DA /r VPMINUB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F DA /r VPMINUB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F DA /r VPMINUB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F38 3A/r VPMINUW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38 3A/r VPMINUW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38 3A/r VPMINUW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Compare packed unsigned word integers in zmm3/m512 and zmm2 and return packed minimum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed unsigned byte or word integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is 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 second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: 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.

Operation

PMINUB (64-bit Operands)

```
    IF DEST[7:0] < SRC[17:0] THEN
        DEST[7:0] := DEST[7:0];
    ELSE
        DEST[7:0] := SRC[7:0]; FI;
    (* Repeat operation for 2nd through 7th bytes in source and destination operands *)
    IF DEST[63:56] < SRC[63:56] THEN
        DEST[63:56] := DEST[63:56];
    ELSE
        DEST[63:56] := SRC[63:56]; FI;
```

PMINUB (128-bit Operands)

```
    IF DEST[7:0] < SRC[7:0] THEN
        DEST[7:0] := DEST[7:0];
    ELSE
        DEST[15:0] := SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] < SRC[127:120] THEN
        DEST[127:120] := DEST[127:120];
    ELSE
        DEST[127:120] := SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINUB (VEX.128 Encoded Version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[7:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] < SRC2[127:120] THEN
        DEST[127:120] := SRC1[127:120];
    ELSE
        DEST[127:120] := SRC2[127:120]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINUB (VEX.256 Encoded Version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[15:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] < SRC2[255:248] THEN
        DEST[255:248] := SRC1[255:248];
    ELSE
        DEST[255:248] := SRC2[255:248]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINUB (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
        IF SRC1[i+7:i] < SRC2[i+7:i]
            THEN DEST[i+7:i] := SRC1[i+7:i];
            ELSE DEST[i+7:i] := SRC2[i+7:i];
        FI;
        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
```

PMINUW (128-bit Operands)

```
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] < SRC[127:112] THEN
        DEST[127:112] := DEST[127:112];
    ELSE
        DEST[127:112] := SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINUW (VEX.128 Encoded Version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] < SRC2[127:112] THEN
        DEST[127:112] := SRC1[127:112];
    ELSE
        DEST[127:112] := SRC2[127:112]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINUW (VEX.256 Encoded Version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] < SRC2[255:240] THEN
        DEST[255:240] := SRC1[255:240];
    ELSE
        DEST[255:240] := SRC2[255:240]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINUW (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
        IF SRC1[i+15:i] < SRC2[i+15:i]
            THEN DEST[i+15:i] := SRC1[i+15:i];
            ELSE DEST[i+15:i] := SRC2[i+15:i];
        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
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMINUB __m512i _mm512_min_epu8( __m512i a, __m512i b);
VPMINUB __m512i _mm512_mask_min_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINUB __m512i _mm512_maskz_min_epu8( __mmask64 k, __m512i a, __m512i b);
VPMINUW __m512i _mm512_min_epu16( __m512i a, __m512i b);
VPMINUW __m512i _mm512_mask_min_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINUW __m512i _mm512_maskz_min_epu16( __mmask32 k, __m512i a, __m512i b);
VPMINUB __m256i _mm256_mask_min_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINUB __m256i _mm256_maskz_min_epu8( __mmask32 k, __m256i a, __m256i b);
VPMINUW __m256i _mm256_mask_min_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUW __m256i _mm256_maskz_min_epu16( __mmask16 k, __m256i a, __m256i b);
VPMINUB __m128i _mm_mask_min_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMINUB __m128i _mm_maskz_min_epu8( __mmask16 k, __m128i a, __m128i b);
VPMINUW __m128i _mm_mask_min_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUW __m128i _mm_maskz_min_epu16( __mmask8 k, __m128i a, __m128i b);
(V)PMINUB __m128i _mm_min_epu8 ( __m128i a, __m128i b)
(V)PMINUW __m128i _mm_min_epu16 ( __m128i a, __m128i b);
VPMINUB __m256i _mm256_min_epu8 ( __m256i a, __m256i b)
VPMINUW __m256i _mm256_min_epu16 ( __m256i a, __m256i b);
PMINUB __m64 _m_min_pu8 (__m64 a, __m64 b)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PMINUD/PMINUQ—Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3B /r PMINUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.128.66.0F38.WIG 3B /r VPMINUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.256.66.0F38.WIG 3B /r VPMINUD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
EVEX.128.66.0F38.W0 3B /r VPMINUD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 3B /r VPMINUD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 3B /r VPMINUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1[1]	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 3B /r VPMINUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 3B /r VPMINUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 3B /r VPMINUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1[1]	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed unsigned dword/qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is 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 second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: 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 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINUD (128-bit Legacy SSE Version)

```
PMINUD instruction for 128-bit operands:
   IF DEST[31:0] < SRC[31:0] THEN
       DEST[31:0] := DEST[31:0];
   ELSE
       DEST[31:0] := SRC[31:0]; FI;
   (* Repeat operation for 2nd through 7th words in source and destination operands *)
   IF DEST[127:96] < SRC[127:96] THEN
       DEST[127:96] := DEST[127:96];
   ELSE
       DEST[127:96] := SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINUD (VEX.128 Encoded Version)

```
VPMINUD instruction for 128-bit operands:
   IF SRC1[31:0] < SRC2[31:0] THEN
       DEST[31:0] := SRC1[31:0];
   ELSE
       DEST[31:0] := SRC2[31:0]; FI;
   (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
   IF SRC1[127:96] < SRC2[127:96] THEN
       DEST[127:96] := SRC1[127:96];
   ELSE
       DEST[127:96] := SRC2[127:96]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINUD (VEX.256 Encoded Version)

```
VPMINUD instruction for 128-bit operands:
   IF SRC1[31:0] < SRC2[31:0] THEN
       DEST[31:0] := SRC1[31:0];
   ELSE
       DEST[31:0] := SRC2[31:0]; FI;
   (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
   IF SRC1[255:224] < SRC2[255:224] THEN
       DEST[255:224] := SRC1[255:224];
   ELSE
       DEST[255:224] := SRC2[255:224]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINUD (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
                IF SRC1[i+31:i] < SRC2[31:0]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] < SRC2[i+31:i]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[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
```

VPMINUQ (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
                IF SRC1[i+63:i] < SRC2[63:0]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] < SRC2[i+63:i]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[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
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINUD __m512i _mm512_min_epu32(__m512i a, __m512i b);
VPMINUD __m512i _mm512_mask_min_epu32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINUD __m512i _mm512_maskz_min_epu32(__mmask16 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_min_epu64(__m512i a, __m512i b);
VPMINUQ __m512i _mm512_mask_min_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_maskz_min_epu64(__mmask8 k, __m512i a, __m512i b);
VPMINUD __m256i _mm256_mask_min_epu32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUD __m256i _mm256_maskz_min_epu32(__mmask16 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_mask_min_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_maskz_min_epu64(__mmask8 k, __m256i a, __m256i b);
VPMINUD __m128i _mm_mask_min_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUD __m128i _mm_maskz_min_epu32(__mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_mask_min_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_maskz_min_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMINUD __m128i _mm_min_epu32 (__m128i a, __m128i b);
VPMINUD __m256i _mm256_min_epu32 (__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PMOVSX—Packed Move With Sign Extend

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 20 /r PMOVSXBW xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
66 0f 38 21 /r PMOVSXBD xmm1, xmm2/m32	A	V/V	SSE4_1	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
66 0f 38 22 /r PMOVSXBQ xmm1, xmm2/m16	A	V/V	SSE4_1	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
66 0f 38 23/r PMOVSXWD xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
66 0f 38 24 /r PMOVSXWQ xmm1, xmm2/m32	A	V/V	SSE4_1	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
66 0f 38 25 /r PMOVSXDQ xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1, xmm2/m64	A	V/V	AVX	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
VEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1, xmm2/m32	A	V/V	AVX	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1, xmm2/m16	A	V/V	AVX	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1, xmm2/m64	A	V/V	AVX	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1, xmm2/m32	A	V/V	AVX	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 25 /r VPMOVSXDQ xmm1, xmm2/m64	A	V/V	AVX	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
VEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1, xmm2/m64	A	V/V	AVX2	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1, xmm2/m32	A	V/V	AVX2	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1, xmm2/m64	A	V/V	AVX2	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 25 /r VPMOVSXDQ ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in ymm1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Sign extend 8 packed 8-bit integers in xmm2/m64 to 8 packed 16-bit integers in zmm1.
EVEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
EVEX.512.66.0F38.WIG 20 /r VPMOVSXBW zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512BW OR AVX10.1[1]	Sign extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.
EVEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1 {k1}{z}, xmm2/m32	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 21 /r VPMOVSXBD zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F OR AVX10.1[1]	Sign extend 16 packed 8-bit integers in the low 16 bytes of xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1 {k1}{z}, xmm2/m16	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1 {k1}{z}, xmm2/m32	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 22 /r VPMOVSXBQ zmm1 {k1}{z}, xmm2/m64	D	V/V	AVX512F OR AVX10.1[1]	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Sign extend 4 packed 16-bit integers in the low 8 bytes of ymm2/mem to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Sign extend 8 packed 16-bit integers in the low 16 bytes of ymm2/m128 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 23 /r VPMOVSXWD zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F OR AVX10.1[1]	Sign extend 16 packed 16-bit integers in the low 32 bytes of ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1 {k1}{z}, xmm2/m32	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 24 /r VPMOVSXWQ zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F OR AVX10.1[1]	Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 25 /r VPMOVSXDQ xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.W0 25 /r VPMOVSXDQ ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 25 /r VPMOVSXDQ zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F OR AVX10.1[1]	Sign extend 8 packed 32-bit integers in the low 32 bytes of ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Legacy and VEX encoded versions: Packed byte, word, or dword integers in the low bytes of the source operand (second operand) are sign extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed byte, word or dword integers starting from the low bytes of the source operand (second operand) are sign extended to word, dword or quadword integers and stored to the destination operand under the writemask. The destination register is XMM, YMM or ZMM Register.

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

Operation

```
Packed_Sign_Extend_BYTE_to_WORD(DEST, SRC)
DEST[15:0] := SignExtend(SRC[7:0]);
DEST[31:16] := SignExtend(SRC[15:8]);
DEST[47:32] := SignExtend(SRC[23:16]);
DEST[63:48] := SignExtend(SRC[31:24]);
DEST[79:64] := SignExtend(SRC[39:32]);
DEST[95:80] := SignExtend(SRC[47:40]);
DEST[111:96] := SignExtend(SRC[55:48]);
DEST[127:112] := SignExtend(SRC[63:56]);
```

Packed_Sign_Extend_BYTE_to_DWORD(DEST, SRC)

```
DEST[31:0] := SignExtend(SRC[7:0]);
DEST[63:32] := SignExtend(SRC[15:8]);
DEST[95:64] := SignExtend(SRC[23:16]);
DEST[127:96] := SignExtend(SRC[31:24]);
```

Packed_Sign_Extend_BYTE_to_QWORD(DEST, SRC)

```
DEST[63:0] := SignExtend(SRC[7:0]);
DEST[127:64] := SignExtend(SRC[15:8]);
```

Packed_Sign_Extend_WORD_to_DWORD(DEST, SRC)

```
DEST[31:0] := SignExtend(SRC[15:0]);
DEST[63:32] := SignExtend(SRC[31:16]);
DEST[95:64] := SignExtend(SRC[47:32]);
DEST[127:96] := SignExtend(SRC[63:48]);
```

Packed_Sign_Extend_WORD_to_QWORD(DEST, SRC)

```
DEST[63:0] := SignExtend(SRC[15:0]);
DEST[127:64] := SignExtend(SRC[31:16]);
```

Packed_Sign_Extend_DWORD_to_QWORD(DEST, SRC)

```
DEST[63:0] := SignExtend(SRC[31:0]);
DEST[127:64] := SignExtend(SRC[63:32]);
```

VPMOVSXBW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TEMP_DEST[i+15:i]
        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] := 0
```

VPMOVSXBD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXBQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXWD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[256:192])
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*           ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXWQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*           ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXBW (VEX.256 Encoded Version)

```
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVSXBD (VEX.256 Encoded Version)

```
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] := 0
```

VPMOVSXBQ (VEX.256 Encoded Version)

```
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
Packed_Sign_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])
DEST[MAXVL-1:256] := 0
```

VPMOVSXWD (VEX.256 Encoded Version)

```
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVSXWQ (VEX.256 Encoded Version)

```
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] := 0
```

VPMOVSXDQ (VEX.256 Encoded Version)

```
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVSXBW (VEX.128 Encoded Version)

```
Packed_Sign_Extend_BYTE_to_WORDDEST[127:0], SRC[127:0]()
DEST[MAXVL-1:128] := 0
```

VPMOVSXBD (VEX.128 Encoded Version)

```
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] := 0
```

VPMOVSXBQ (VEX.128 Encoded Version)

```
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] := 0
```

VPMOVSXWD (VEX.128 Encoded Version)

```
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] := 0
```

VPMOVSXWQ (VEX.128 Encoded Version)

```
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] := 0
```

VPMOVSXDQ (VEX.128 Encoded Version)

```
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] := 0
```

PMOVSXBW

```
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

PMOVSXBD

```
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

PMOVSXBQ

```
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

PMOVSXWD

```
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

PMOVSXWQ

```
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

PMOVSXDQ

```
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMOVSXBW __m512i _mm512_cvtepi8_epi16(__m512i a);
VPMOVSXBW __m512i _mm512_mask_cvtepi8_epi16(__m512i a, __mmask32 k, __m512i b);
VPMOVSXBW __m512i _mm512_maskz_cvtepi8_epi16( __mmask32 k, __m512i b);
VPMOVSXBD __m512i _mm512_cvtepi8_epi32(__m512i a);
```

VPMOVSXBD __m512i _mm512_mask_cvtepi8_epi32(__m512i a, __mmask16 k, __m512i b);
VPMOVSXBD __m512i _mm512_maskz_cvtepi8_epi32(__mmask16 k, __m512i b);
VPMOVSXBQ __m512i _mm512_cvtepi8_epi64(__m512i a);
VPMOVSXBQ __m512i _mm512_mask_cvtepi8_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVSXBQ __m512i _mm512_maskz_cvtepi8_epi64(__mmask8 k, __m512i a);
VPMOVSXDQ __m512i _mm512_cvtepi32_epi64(__m512i a);
VPMOVSXDQ __m512i _mm512_mask_cvtepi32_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVSXDQ __m512i _mm512_maskz_cvtepi32_epi64(__mmask8 k, __m512i a);
VPMOVSXWD __m512i _mm512_cvtepi16_epi32(__m512i a);
VPMOVSXWD __m512i _mm512_mask_cvtepi16_epi32(__m512i a, __mmask16 k, __m512i b);
VPMOVSXWD __m512i _mm512_maskz_cvtepi16_epi32(__mmask16 k, __m512i a);
VPMOVSXWQ __m512i _mm512_cvtepi16_epi64(__m512i a);
VPMOVSXWQ __m512i _mm512_mask_cvtepi16_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVSXWQ __m512i _mm512_maskz_cvtepi16_epi64(__mmask8 k, __m512i a);
VPMOVSXBW __m256i _mm256_cvtepi8_epi16(__m256i a);
VPMOVSXBW __m256i _mm256_mask_cvtepi8_epi16(__m256i a, __mmask16 k, __m256i b);
VPMOVSXBW __m256i _mm256_maskz_cvtepi8_epi16(__mmask16 k, __m256i b);
VPMOVSXBD __m256i _mm256_cvtepi8_epi32(__m256i a);
VPMOVSXBD __m256i _mm256_mask_cvtepi8_epi32(__m256i a, __mmask8 k, __m256i b);
VPMOVSXBD __m256i _mm256_maskz_cvtepi8_epi32(__mmask8 k, __m256i b);
VPMOVSXBQ __m256i _mm256_cvtepi8_epi64(__m256i a);
VPMOVSXBQ __m256i _mm256_mask_cvtepi8_epi64(__m256i a, __mmask8 k, __m256i b);
VPMOVSXBQ __m256i _mm256_maskz_cvtepi8_epi64(__mmask8 k, __m256i a);
VPMOVSXDQ __m256i _mm256_cvtepi32_epi64(__m256i a);
VPMOVSXDQ __m256i _mm256_mask_cvtepi32_epi64(__m256i a, __mmask8 k, __m256i b);
VPMOVSXDQ __m256i _mm256_maskz_cvtepi32_epi64(__mmask8 k, __m256i a);
VPMOVSXWD __m256i _mm256_cvtepi16_epi32(__m256i a);
VPMOVSXWD __m256i _mm256_mask_cvtepi16_epi32(__m256i a, __mmask16 k, __m256i b);
VPMOVSXWD __m256i _mm256_maskz_cvtepi16_epi32(__mmask16 k, __m256i a);
VPMOVSXWQ __m256i _mm256_cvtepi16_epi64(__m256i a);
VPMOVSXWQ __m256i _mm256_mask_cvtepi16_epi64(__m256i a, __mmask8 k, __m256i b);
VPMOVSXWQ __m256i _mm256_maskz_cvtepi16_epi64(__mmask8 k, __m256i a);
VPMOVSXBW __m128i _mm_mask_cvtepi8_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVSXBW __m128i _mm_maskz_cvtepi8_epi16(__mmask8 k, __m128i b);
VPMOVSXBD __m128i _mm_mask_cvtepi8_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSXBD __m128i _mm_maskz_cvtepi8_epi32(__mmask8 k, __m128i b);
VPMOVSXBQ __m128i _mm_mask_cvtepi8_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVSXBQ __m128i _mm_maskz_cvtepi8_epi64(__mmask8 k, __m128i a);
VPMOVSXDQ __m128i _mm_mask_cvtepi32_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVSXDQ __m128i _mm_maskz_cvtepi32_epi64(__mmask8 k, __m128i a);
VPMOVSXWD __m128i _mm_mask_cvtepi16_epi32(__m128i a, __mmask16 k, __m128i b);
VPMOVSXWD __m128i _mm_maskz_cvtepi16_epi32(__mmask16 k, __m128i a);
VPMOVSXWQ __m128i _mm_mask_cvtepi16_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVSXWQ __m128i _mm_maskz_cvtepi16_epi64(__mmask8 k, __m128i a);
PMOVSXBW __m128i _mm_ cvtepi8_epi16 (__m128i a);
PMOVSXBD __m128i _mm_ cvtepi8_epi32 (__m128i a);
PMOVSXBQ __m128i _mm_ cvtepi8_epi64 (__m128i a);
PMOVSXWD __m128i _mm_ cvtepi16_epi32 (__m128i a);
PMOVSXWQ __m128i _mm_ cvtepi16_epi64 (__m128i a);
PMOVSXDQ __m128i _mm_ cvtepi32_epi64 (__m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

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

Additionally:

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

PMOVZX—Packed Move With Zero Extend

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 30 /r PMOVZXBW xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
66 0f 38 31 /r PMOVZXBD xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
66 0f 38 32 /r PMOVZXBQ xmm1, xmm2/m16	A	V/V	SSE4_1	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
66 0f 38 33 /r PMOVZXWD xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
66 0f 38 34 /r PMOVZXWQ xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
66 0f 38 35 /r PMOVZXDQ xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 30 /r VPMOVZXBW xmm1, xmm2/m64	A	V/V	AVX	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
VEX.128.66.0F38.WIG 31 /r VPMOVZXBD xmm1, xmm2/m32	A	V/V	AVX	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 32 /r VPMOVZXBQ xmm1, xmm2/m16	A	V/V	AVX	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 33 /r VPMOVZXWD xmm1, xmm2/m64	A	V/V	AVX	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 34 /r VPMOVZXWQ xmm1, xmm2/m32	A	V/V	AVX	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F 38.WIG 35 /r VPMOVZXDQ xmm1, xmm2/m64	A	V/V	AVX	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
VEX.256.66.0F38.WIG 31 /r VPMOVZXBD ymm1, xmm2/m64	A	V/V	AVX2	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 32 /r VPMOVZXBQ ymm1, xmm2/m32	A	V/V	AVX2	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 33 /r VPMOVZXWD ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 8 packed 16-bit integers xmm2/m128 to 8 packed 32-bit integers in ymm1.

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
VEX.256.66.0F38.WIG 34 /r VPMOVZXWQ ymm1, xmm2/m64	A	V/V	AVX2	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 35 /r VPMOVZXDQ ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in ymm1.
EVEX.128.66.0F38 30.WIG /r VPMOVZXBW xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
EVEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
EVEX.512.66.0F38.WIG 30 /r VPMOVZXBW zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512BW OR AVX10.1[1]	Zero extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.
EVEX.128.66.0F38.WIG 31 /r VPMOVZXBD xmm1 {k1}{z}, xmm2/m32	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 31 /r VPMOVZXBD ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 31 /r VPMOVZXBD zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F OR AVX10.1[1]	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 32 /r VPMOVZXBQ xmm1 {k1}{z}, xmm2/m16	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 32 /r VPMOVZXBQ ymm1 {k1}{z}, xmm2/m32	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 32 /r VPMOVZXBQ zmm1 {k1}{z}, xmm2/m64	D	V/V	AVX512F OR AVX10.1[1]	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 33 /r VPMOVZXWD xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 33 /r VPMOVZXWD ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 33 /r VPMOVZXWD zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F OR AVX10.1[1]	Zero extend 16 packed 16-bit integers in ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 34 /r VPMOVZXWQ xmm1 {k1}{z}, xmm2/m32	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 34 /r VPMOVZXWQ ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.WIG 34 /r VPMOVZXWQ zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F OR AVX10.1[1]	Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 35 /r VPMOVZXDQ xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 35 /r VPMOVZXDQ ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 35 /r VPMOVZXDQ zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F OR AVX10.1[1]	Zero extend 8 packed 32-bit integers in ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Legacy, VEX, and EVEX encoded versions: Packed byte, word, or dword integers starting from the low bytes of the source operand (second operand) are zero extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

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

EVEX encoded versions: Packed dword integers starting from the low bytes of the source operand (second operand) are zero extended to quadword integers and stored to the destination operand under the writemask.The destination register is XMM, YMM or ZMM Register.

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

Operation

```
Packed_Zero_Extend_BYTE_to_WORD(DEST, SRC)
DEST[15:0] := ZeroExtend(SRC[7:0]);
DEST[31:16] := ZeroExtend(SRC[15:8]);
DEST[47:32] := ZeroExtend(SRC[23:16]);
DEST[63:48] := ZeroExtend(SRC[31:24]);
DEST[79:64] := ZeroExtend(SRC[39:32]);
DEST[95:80] := ZeroExtend(SRC[47:40]);
DEST[111:96] := ZeroExtend(SRC[55:48]);
DEST[127:112] := ZeroExtend(SRC[63:56]);
```

Packed_Zero_Extend_BYTE_to_DWORD(DEST, SRC)

```
DEST[31:0] := ZeroExtend(SRC[7:0]);
DEST[63:32] := ZeroExtend(SRC[15:8]);
DEST[95:64] := ZeroExtend(SRC[23:16]);
DEST[127:96] := ZeroExtend(SRC[31:24]);
```

Packed_Zero_Extend_BYTE_to_QWORD(DEST, SRC)

```
DEST[63:0] := ZeroExtend(SRC[7:0]);
DEST[127:64] := ZeroExtend(SRC[15:8]);
```

Packed_Zero_Extend_WORD_to_DWORD(DEST, SRC)

```
DEST[31:0] := ZeroExtend(SRC[15:0]);
DEST[63:32] := ZeroExtend(SRC[31:16]);
DEST[95:64] := ZeroExtend(SRC[47:32]);
DEST[127:96] := ZeroExtend(SRC[63:48]);
```

Packed_Zero_Extend_WORD_to_QWORD(DEST, SRC)

```
DEST[63:0] := ZeroExtend(SRC[15:0]);
DEST[127:64] := ZeroExtend(SRC[31:16]);
```

Packed_Zero_Extend_DWORD_to_QWORD(DEST, SRC)

```
DEST[63:0] := ZeroExtend(SRC[31:0]);
DEST[127:64] := ZeroExtend(SRC[63:32]);
```

VPMOVZXBW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TEMP_DEST[i+15:i]
        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] := 0
```

VPMOVZXBD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXBQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXWD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[256:192])
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXWQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXBW (VEX.256 Encoded Version)

```
Packed_Zero_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVZXBD (VEX.256 Encoded Version)

```
Packed_Zero_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] := 0
```

VPMOVZXBQ (VEX.256 Encoded Version)

```
Packed_Zero_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
Packed_Zero_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])
DEST[MAXVL-1:256] := 0
```

VPMOVZXWD (VEX.256 Encoded Version)

```
Packed_Zero_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVZXWQ (VEX.256 Encoded Version)

```
Packed_Zero_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] := 0
```

VPMOVZXDQ (VEX.256 Encoded Version)

```
Packed_Zero_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVZXBW (VEX.128 Encoded Version)

```
Packed_Zero_Extend_BYTE_to_WORD()
DEST[MAXVL-1:128] := 0
```

VPMOVZXBD (VEX.128 Encoded Version)

```
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[MAXVL-1:128] := 0
```

VPMOVZXBQ (VEX.128 Encoded Version)

```
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[MAXVL-1:128] := 0
```

VPMOVZXWD (VEX.128 Encoded Version)

```
Packed_Zero_Extend_WORD_to_DWORD()
DEST[MAXVL-1:128] := 0
```

VPMOVZXWQ (VEX.128 Encoded Version)

```
Packed_Zero_Extend_WORD_to_QWORD()
DEST[MAXVL-1:128] := 0
```

VPMOVZXDQ (VEX.128 Encoded Version)

```
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[MAXVL-1:128] := 0
```

PMOVZXBW

```
Packed_Zero_Extend_BYTE_to_WORD()
DEST[MAXVL-1:128] (Unmodified)
```

PMOVZXBD

```
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[MAXVL-1:128] (Unmodified)
```

PMOVZXBQ

```
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)
```

PMOVZXWD

```
Packed_Zero_Extend_WORD_to_DWORD()
DEST[MAXVL-1:128] (Unmodified)
```

PMOVZXWQ

```
Packed_Zero_Extend_WORD_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)
```

PMOVZXDQ

```
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMOVZXBW __m512i _mm512_cvtepu8_epi16(__m256i a);
VPMOVZXBW __m512i _mm512_mask_cvtepu8_epi16(__m512i a, __mmask32 k, __m256i b);
VPMOVZXBW __m512i _mm512_maskz_cvtepu8_epi16( __mmask32 k, __m256i b);
VPMOVZXBD __m512i _mm512_cvtepu8_epi32(__m128i a);
```

VPMOVZXBD __m512i _mm512_mask_cvtepu8_epi32(__m512i a, __mmask16 k, __m128i b);
VPMOVZXBD __m512i _mm512_maskz_cvtepu8_epi32(__mmask16 k, __m128i b);
VPMOVZXBQ __m512i _mm512_cvtepu8_epi64(__m128i a);
VPMOVZXBQ __m512i _mm512_mask_cvtepu8_epi64(__m512i a, __mmask8 k, __m128i b);
VPMOVZXBQ __m512i _mm512_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);
VPMOVZXDQ __m512i _mm512_cvtepu32_epi64(__m256i a);
VPMOVZXDQ __m512i _mm512_mask_cvtepu32_epi64(__m512i a, __mmask8 k, __m256i b);
VPMOVZXDQ __m512i _mm512_maskz_cvtepu32_epi64(__mmask8 k, __m256i a);
VPMOVZXWD __m512i _mm512_cvtepu16_epi32(__m128i a);
VPMOVZXWD __m512i _mm512_mask_cvtepu16_epi32(__m512i a, __mmask16 k, __m128i b);
VPMOVZXWD __m512i _mm512_maskz_cvtepu16_epi32(__mmask16 k, __m128i a);
VPMOVZXWQ __m512i _mm512_cvtepu16_epi64(__m256i a);
VPMOVZXWQ __m512i _mm512_mask_cvtepu16_epi64(__m512i a, __mmask8 k, __m256i b);
VPMOVZXWQ __m512i _mm512_maskz_cvtepu16_epi64(__mmask8 k, __m256i a);
VPMOVZXBW __m256i _mm256_cvtepu8_epi16(__m256i a);
VPMOVZXBW __m256i _mm256_mask_cvtepu8_epi16(__m256i a, __mmask16 k, __m128i b);
VPMOVZXBW __m256i _mm256_maskz_cvtepu8_epi16(__mmask16 k, __m128i b);
VPMOVZXBD __m256i _mm256_cvtepu8_epi32(__m128i a);
VPMOVZXBD __m256i _mm256_mask_cvtepu8_epi32(__m256i a, __mmask8 k, __m128i b);
VPMOVZXBD __m256i _mm256_maskz_cvtepu8_epi32(__mmask8 k, __m128i b);
VPMOVZXBQ __m256i _mm256_cvtepu8_epi64(__m128i a);
VPMOVZXBQ __m256i _mm256_mask_cvtepu8_epi64(__m256i a, __mmask8 k, __m128i b);
VPMOVZXBQ __m256i _mm256_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);
VPMOVZXDQ __m256i _mm256_cvtepu32_epi64(__m128i a);
VPMOVZXDQ __m256i _mm256_mask_cvtepu32_epi64(__m256i a, __mmask8 k, __m128i b);
VPMOVZXDQ __m256i _mm256_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);
VPMOVZXWD __m256i _mm256_cvtepu16_epi32(__m128i a);
VPMOVZXWD __m256i _mm256_mask_cvtepu16_epi32(__m256i a, __mmask16 k, __m128i b);
VPMOVZXWD __m256i _mm256_maskz_cvtepu16_epi32(__mmask16 k, __m128i a);
VPMOVZXWQ __m256i _mm256_cvtepu16_epi64(__m128i a);
VPMOVZXWQ __m256i _mm256_mask_cvtepu16_epi64(__m256i a, __mmask8 k, __m128i b);
VPMOVZXWQ __m256i _mm256_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);
VPMOVZXBW __m128i _mm_mask_cvtepu8_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVZXBW __m128i _mm_maskz_cvtepu8_epi16(__mmask8 k, __m128i b);
VPMOVZXBD __m128i _mm_mask_cvtepu8_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi32(__mmask8 k, __m128i b);
VPMOVZXBQ __m128i _mm_mask_cvtepu8_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVZXBQ __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);
VPMOVZXDQ __m128i _mm_mask_cvtepu32_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVZXDQ __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);
VPMOVZXWD __m128i _mm_mask_cvtepu16_epi32(__m128i a, __mmask16 k, __m128i b);
VPMOVZXWD __m128i _mm_maskz_cvtepu16_epi32(__mmask8 k, __m128i a);
VPMOVZXWQ __m128i _mm_mask_cvtepu16_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVZXWQ __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);
PMOVZXBW __m128i _mm_ cvtepu8_epi16 (__m128i a);
PMOVZXBD __m128i _mm_ cvtepu8_epi32 (__m128i a);
PMOVZXBQ __m128i _mm_ cvtepu8_epi64 (__m128i a);
PMOVZXWD __m128i _mm_ cvtepu16_epi32 (__m128i a);
PMOVZXWQ __m128i _mm_ cvtepu16_epi64 (__m128i a);
PMOVZXDQ __m128i _mm_ cvtepu32_epi64 (__m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

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

Additionally:

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

PMULDQ—Multiply Packed Doubleword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 28 /r PMULDQ xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply packed signed doubleword integers in xmm1 by packed signed doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.128.66.0F38.WIG 28 /r VPMULDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.256.66.0F38.WIG 28 /r VPMULDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256, and store the quadword results in ymm1.
EVEX.128.66.0F38.W1 28 /r VPMULDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 28 /r VPMULDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 28 /r VPMULDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1[1]	Multiply packed signed doubleword integers in zmm2 by packed signed doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Multiplies packed signed doubleword integers in the even-numbered (zero-based reference) elements of the first source operand with the packed signed doubleword integers in the corresponding elements of the second source operand and stores packed signed quadword results in the destination operand.

128-bit Legacy SSE version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination XMM operand is the same. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first, 3rd, 5th, 7th doubleword element. For 256-bit memory operands, 256 bits are fetched from memory, but only the four even-numbered doublewords are used in the computation. The first source operand and the destination operand are YMM registers. The second source operand can be a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an 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 is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

Operation

VPMULDQ (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] := SignExtend64( SRC1[i+31:i]) * SignExtend64( SRC2[31:0])
                ELSE DEST[i+63:i] := SignExtend64( SRC1[i+31:i]) * SignExtend64( SRC2[i+31: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
```

VPMULDQ (VEX.256 Encoded Version)

```
DEST[63:0] := SignExtend64( SRC1[31:0]) * SignExtend64( SRC2[31:0])
DEST[127:64] := SignExtend64( SRC1[95:64]) * SignExtend64( SRC2[95:64])
DEST[191:128] := SignExtend64( SRC1[159:128]) * SignExtend64( SRC2[159:128])
DEST[255:192] := SignExtend64( SRC1[223:192]) * SignExtend64( SRC2[223:192])
DEST[MAXVL-1:256] := 0
```

VPMULDQ (VEX.128 Encoded Version)

```
DEST[63:0] := SignExtend64( SRC1[31:0]) * SignExtend64( SRC2[31:0])
DEST[127:64] := SignExtend64( SRC1[95:64]) * SignExtend64( SRC2[95:64])
DEST[MAXVL-1:128] := 0
```

PMULDQ (128-bit Legacy SSE Version)

```
DEST[63:0] := SignExtend64( DEST[31:0]) * SignExtend64( SRC[31:0])
DEST[127:64] := SignExtend64( DEST[95:64]) * SignExtend64( SRC[95:64])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULDQ __m512i _mm512_mul_epi32(__m512i a, __m512i b);
VPMULDQ __m512i _mm512_mask_mul_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULDQ __m512i _mm512_maskz_mul_epi32( __mmask8 k, __m512i a, __m512i b);
```

VPMULDQ __m256i _mm256_mask_mul_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULDQ __m256i _mm256_mask_mul_epi32(__mmask8 k, __m256i a, __m256i b);
VPMULDQ __m128i _mm_mask_mul_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULDQ __m128i _mm_mask_mul_epi32(__mmask8 k, __m128i a, __m128i b);
(V)PMULDQ __m128i _mm_mul_epi32(__m128i a, __m128i b);
VPMULDQ __m256i _mm256_mul_epi32(__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PMULHRSW—Packed Multiply High With Round and Scale

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 0B /r[1] PMULHRSW mm1, mm2/m64	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to mm1.
66 0F 38 0B /r PMULHRSW xmm1, xmm2/m128	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1.
VEX.128.66.0F38.WIG 0B /r VPMULHRSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1.
VEX.256.66.0F38.WIG 0B /r VPMULHRSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to ymm1.
EVEX.128.66.0F38.WIG 0B /r VPMULHRSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 0B /r VPMULHRSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 0B /r VPMULHRSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

PMULHRSW multiplies vertically each signed 16-bit integer from the destination operand (first operand) with the corresponding signed 16-bit integer of the source operand (second operand), producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.

When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: 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 conditionally updated with writemask k1.

Operation

PMULHRSW (With 64-bit Operands)

```
    temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
    temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
    temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >> 14) + 1;
    temp3[31:0] = INT32 ((DEST[63:48] * SRc[63:48]) >> 14) + 1;
    DEST[15:0] = temp0[16:1];
    DEST[31:16] = temp1[16:1];
    DEST[47:32] = temp2[16:1];
    DEST[63:48] = temp3[16:1];
```

PMULHRSW (With 128-bit Operands)

```
    temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
    temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
    temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1;
    temp3[31:0] = INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1;
    temp4[31:0] = INT32 ((DEST[79:64] * SRC[79:64]) >>14) + 1;
    temp5[31:0] = INT32 ((DEST[95:80] * SRC[95:80]) >>14) + 1;
    temp6[31:0] = INT32 ((DEST[111:96] * SRC[111:96]) >>14) + 1;
    temp7[31:0] = INT32 ((DEST[127:112] * SRC[127:112) >>14) + 1;
    DEST[15:0] = temp0[16:1];
    DEST[31:16] = temp1[16:1];
    DEST[47:32] = temp2[16:1];
    DEST[63:48] = temp3[16:1];
    DEST[79:64] = temp4[16:1];
    DEST[95:80] = temp5[16:1];
    DEST[111:96] = temp6[16:1];
    DEST[127:112] = temp7[16:1];
```

VPMULHRSW (VEX.128 Encoded Version)

```
temp0[31:0] := INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] := INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] := INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] := INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] := INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] := INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] := INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] := INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
DEST[15:0] := temp0[16:1]
```

```
DEST[31:16] := temp1[16:1]
DEST[47:32] := temp2[16:1]
DEST[63:48] := temp3[16:1]
DEST[79:64] := temp4[16:1]
DEST[95:80] := temp5[16:1]
DEST[111:96] := temp6[16:1]
DEST[127:112] := temp7[16:1]
DEST[MAXVL-1:128] := 0
```

VPMULHRSW (VEX.256 Encoded Version)

```
temp0[31:0] := INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] := INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] := INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] := INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] := INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] := INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] := INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] := INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
temp8[31:0] := INT32 ((SRC1[143:128] * SRC2[143:128]) >>14) + 1
temp9[31:0] := INT32 ((SRC1[159:144] * SRC2[159:144]) >>14) + 1
temp10[31:0] := INT32 ((SRC1[75:160] * SRC2[175:160]) >>14) + 1
temp11[31:0] := INT32 ((SRC1[191:176] * SRC2[191:176]) >>14) + 1
temp12[31:0] := INT32 ((SRC1[207:192] * SRC2[207:192]) >>14) + 1
temp13[31:0] := INT32 ((SRC1[223:208] * SRC2[223:208]) >>14) + 1
temp14[31:0] := INT32 ((SRC1[239:224] * SRC2[239:224]) >>14) + 1
temp15[31:0] := INT32 ((SRC1[255:240] * SRC2[255:240) >>14) + 1

DEST[15:0] := temp0[16:1]
DEST[31:16] := temp1[16:1]
DEST[47:32] := temp2[16:1]
DEST[63:48] := temp3[16:1]
DEST[79:64] := temp4[16:1]
DEST[95:80] := temp5[16:1]
DEST[111:96] := temp6[16:1]
DEST[127:112] := temp7[16:1]
DEST[143:128] := temp8[16:1]
DEST[159:144] := temp9[16:1]
DEST[175:160] := temp10[16:1]
DEST[191:176] := temp11[16:1]
DEST[207:192] := temp12[16:1]
DEST[223:208] := temp13[16:1]
DEST[239:224] := temp14[16:1]
DEST[255:240] := temp15[16:1]
DEST[MAXVL-1:256] := 0
```

VPMULHRSW (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
            temp[31:0] := ((SRC1[i+15:i] * SRC2[i+15:i]) >>14) + 1
            DEST[i+15:i] := tmp[16:1]
        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] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMULHRSW __m512i _mm512_mulhrs_epi16(__m512i a, __m512i b);
VPMULHRSW __m512i _mm512_mask_mulhrs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m512i _mm512_maskz_mulhrs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m256i _mm256_mask_mulhrs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m256i _mm256_maskz_mulhrs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m128i _mm_mask_mulhrs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHRSW __m128i _mm_maskz_mulhrs_epi16(__mmask8 k, __m128i a, __m128i b);
PMULHRSW __m64 _mm_mulhrs_pi16 (__m64 a, __m64 b)
(V)PMULHRSW __m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)
VPMULHRSW __m256i _mm256_mulhrs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PMULHUW—Multiply Packed Unsigned Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E4 /r[1] PMULHUW mm1, mm2/m64	A	V/V	SSE	Multiply the packed unsigned word integers in mm1 register and mm2/m64, and store the high 16 bits of the results in mm1.
66 0F E4 /r PMULHUW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed unsigned word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.
VEX.128.66.0F.WIG E4 /r VPMULHUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.
VEX.256.66.0F.WIG E4 /r VPMULHUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1.
EVEX.128.66.0F.WIG E4 /r VPMULHUW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG E4 /r VPMULHUW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG E4 /r VPMULHUW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Multiply the packed unsigned word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD unsigned multiply of the packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 1-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: 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 conditionally updated with writemask k1.

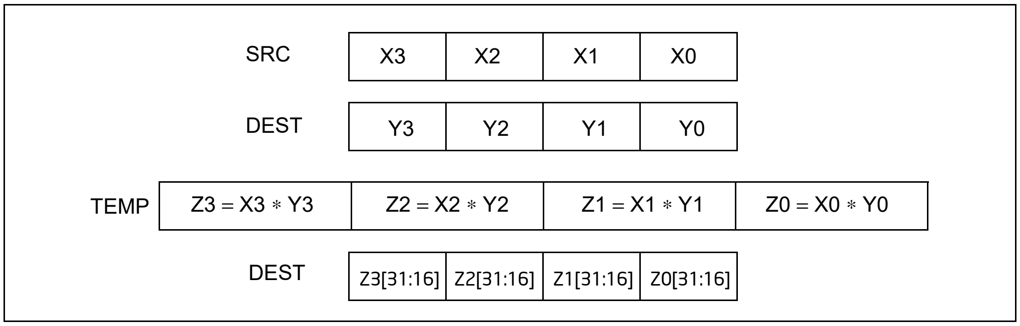


Figure 1-12. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

Operation

PMULHUW (With 64-bit Operands)

```
    TEMP0[31:0] := DEST[15:0] ∗ SRC[15:0]; (* Unsigned multiplication *)
    TEMP1[31:0] := DEST[31:16] ∗ SRC[31:16];
    TEMP2[31:0] := DEST[47:32] ∗ SRC[47:32];
    TEMP3[31:0] := DEST[63:48] ∗ SRC[63:48];
    DEST[15:0] := TEMP0[31:16];
    DEST[31:16] := TEMP1[31:16];
    DEST[47:32] := TEMP2[31:16];
    DEST[63:48] := TEMP3[31:16];
```

PMULHUW (With 128-bit Operands)

```
    TEMP0[31:0] := DEST[15:0] ∗ SRC[15:0]; (* Unsigned multiplication *)
    TEMP1[31:0] := DEST[31:16] ∗ SRC[31:16];
    TEMP2[31:0] := DEST[47:32] ∗ SRC[47:32];
    TEMP3[31:0] := DEST[63:48] ∗ SRC[63:48];
    TEMP4[31:0] := DEST[79:64] ∗ SRC[79:64];
    TEMP5[31:0] := DEST[95:80] ∗ SRC[95:80];
    TEMP6[31:0] := DEST[111:96] ∗ SRC[111:96];
    TEMP7[31:0] := DEST[127:112] ∗ SRC[127:112];
    DEST[15:0] := TEMP0[31:16];
    DEST[31:16] := TEMP1[31:16];
    DEST[47:32] := TEMP2[31:16];
    DEST[63:48] := TEMP3[31:16];
    DEST[79:64] := TEMP4[31:16];
    DEST[95:80] := TEMP5[31:16];
    DEST[111:96] := TEMP6[31:16];
```

```
    DEST[127:112] := TEMP7[31:16];
```

VPMULHUW (VEX.128 Encoded Version)

```
TEMP0[31:0] := SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] := SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] := SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] := SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] := SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] := SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] := SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] := SRC1[127:112] * SRC2[127:112]
DEST[15:0] := TEMP0[31:16]
DEST[31:16] := TEMP1[31:16]
DEST[47:32] := TEMP2[31:16]
DEST[63:48] := TEMP3[31:16]
DEST[79:64] := TEMP4[31:16]
DEST[95:80] := TEMP5[31:16]
DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[MAXVL-1:128] := 0
```

PMULHUW (VEX.256 Encoded Version)

```
TEMP0[31:0] := SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] := SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] := SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] := SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] := SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] := SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] := SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] := SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] := SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] := SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] := SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] := SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] := SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] := SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] := SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] := SRC1[255:240] * SRC2[255:240]
DEST[15:0] := TEMP0[31:16]
DEST[31:16] := TEMP1[31:16]
DEST[47:32] := TEMP2[31:16]
DEST[63:48] := TEMP3[31:16]
DEST[79:64] := TEMP4[31:16]
DEST[95:80] := TEMP5[31:16]
DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[143:128] := TEMP8[31:16]
DEST[159:144] := TEMP9[31:16]
DEST[175:160] := TEMP10[31:16]
DEST[191:176] := TEMP11[31:16]
DEST[207:192] := TEMP12[31:16]
DEST[223:208] := TEMP13[31:16]
DEST[239:224] := TEMP14[31:16]
DEST[255:240] := TEMP15[31:16]
```

```
DEST[MAXVL-1:256] := 0
```

PMULHUW (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
            temp[31:0] := SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] := tmp[31:16]
        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] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULHUW __m512i _mm512_mulhi_epu16(__m512i a, __m512i b);
VPMULHUW __m512i _mm512_mask_mulhi_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHUW __m512i _mm512_maskz_mulhi_epu16( __mmask32 k, __m512i a, __m512i b);
VPMULHUW __m256i _mm256_mask_mulhi_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHUW __m256i _mm256_maskz_mulhi_epu16( __mmask16 k, __m256i a, __m256i b);
VPMULHUW __m128i _mm_mask_mulhi_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHUW __m128i _mm_maskz_mulhi_epu16( __mmask8 k, __m128i a, __m128i b);
PMULHUW __m64 _mm_mulhi_pu16(__m64 a, __m64 b)
(V)PMULHUW __m128i _mm_mulhi_epu16 ( __m128i a, __m128i b)
VPMULHUW __m256i _mm256_mulhi_epu16 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PMULHW—Multiply Packed Signed Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E5 /r[1] PMULHW mm, mm/m64	A	V/V	MMX	Multiply the packed signed word integers in mm1 register and mm2/m64, and store the high 16 bits of the results in mm1.
66 0F E5 /r PMULHW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.
VEX.128.66.0F.WIG E5 /r VPMULHW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.
VEX.256.66.0F.WIG E5 /r VPMULHW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1.
EVEX.128.66.0F.WIG E5 /r VPMULHW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG E5 /r VPMULHW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG E5 /r VPMULHW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 1-12 shows this operation when using 64-bit operands.)

n 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: 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 conditionally updated with writemask k1.

Operation

PMULHW (With 64-bit Operands)

```
    TEMP0[31:0] := DEST[15:0] ∗ SRC[15:0]; (* Signed multiplication *)
    TEMP1[31:0] := DEST[31:16] ∗ SRC[31:16];
    TEMP2[31:0] := DEST[47:32] ∗ SRC[47:32];
    TEMP3[31:0] := DEST[63:48] ∗ SRC[63:48];
    DEST[15:0] := TEMP0[31:16];
    DEST[31:16] := TEMP1[31:16];
    DEST[47:32] := TEMP2[31:16];
    DEST[63:48] := TEMP3[31:16];
```

PMULHW (With 128-bit Operands)

```
    TEMP0[31:0] := DEST[15:0] ∗ SRC[15:0]; (* Signed multiplication *)
    TEMP1[31:0] := DEST[31:16] ∗ SRC[31:16];
    TEMP2[31:0] := DEST[47:32] ∗ SRC[47:32];
    TEMP3[31:0] := DEST[63:48] ∗ SRC[63:48];
    TEMP4[31:0] := DEST[79:64] ∗ SRC[79:64];
    TEMP5[31:0] := DEST[95:80] ∗ SRC[95:80];
    TEMP6[31:0] := DEST[111:96] ∗ SRC[111:96];
    TEMP7[31:0] := DEST[127:112] ∗ SRC[127:112];
    DEST[15:0] := TEMP0[31:16];
    DEST[31:16] := TEMP1[31:16];
    DEST[47:32] := TEMP2[31:16];
    DEST[63:48] := TEMP3[31:16];
    DEST[79:64] := TEMP4[31:16];
    DEST[95:80] := TEMP5[31:16];
    DEST[111:96] := TEMP6[31:16];
    DEST[127:112] := TEMP7[31:16];
```

VPMULHW (VEX.128 Encoded Version)

```
TEMP0[31:0] := SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] := SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] := SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] := SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] := SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] := SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] := SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] := SRC1[127:112] * SRC2[127:112]
DEST[15:0] := TEMP0[31:16]
DEST[31:16] := TEMP1[31:16]
DEST[47:32] := TEMP2[31:16]
```

```
DEST[63:48] := TEMP3[31:16]
DEST[79:64] := TEMP4[31:16]
DEST[95:80] := TEMP5[31:16]
DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[MAXVL-1:128] := 0
```

PMULHW (VEX.256 Encoded Version)

```
TEMP0[31:0] := SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] := SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] := SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] := SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] := SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] := SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] := SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] := SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] := SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] := SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] := SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] := SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] := SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] := SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] := SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] := SRC1[255:240] * SRC2[255:240]
DEST[15:0] := TEMP0[31:16]
DEST[31:16] := TEMP1[31:16]
DEST[47:32] := TEMP2[31:16]
DEST[63:48] := TEMP3[31:16]
DEST[79:64] := TEMP4[31:16]
DEST[95:80] := TEMP5[31:16]
DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[143:128] := TEMP8[31:16]
DEST[159:144] := TEMP9[31:16]
DEST[175:160] := TEMP10[31:16]
DEST[191:176] := TEMP11[31:16]
DEST[207:192] := TEMP12[31:16]
DEST[223:208] := TEMP13[31:16]
DEST[239:224] := TEMP14[31:16]
DEST[255:240] := TEMP15[31:16]
DEST[MAXVL-1:256] := 0
```

PMULHW (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
            temp[31:0] := SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] := tmp[31:16]
        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] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULHW __m512i _mm512_mulhi_epi16(__m512i a, __m512i b);
VPMULHW __m512i _mm512_mask_mulhi_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHW __m512i _mm512_maskz_mulhi_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULHW __m256i _mm256_mask_mulhi_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHW __m256i _mm256_maskz_mulhi_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHW __m128i _mm_mask_mulhi_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHW __m128i _mm_maskz_mulhi_epi16(__mmask8 k, __m128i a, __m128i b);
PMULHW __m64 _mm_mulhi_pi16 (__m64 m1, __m64 m2)
(V)PMULHW __m128i _mm_mulhi_epi16 (__m128i a, __m128i b)
VPMULHW __m256i _mm256_mulhi_epi16 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PMULLD/PMULLQ—Multiply Packed Integers and Store Low Result

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 40 /r PMULLD xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply the packed dword signed integers in xmm1 and xmm2/m128 and store the low 32 bits of each product in xmm1.
VEX.128.66.0F38.WIG 40 /r VPMULLD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.
VEX.256.66.0F38.WIG 40 /r VPMULLD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed dword signed integers in ymm2 and ymm3/m256 and store the low 32 bits of each product in ymm1.
EVEX.128.66.0F38.W0 40 /r VPMULLD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Multiply the packed dword signed integers in xmm2 and xmm3/m128/m32bcst and store the low 32 bits of each product in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 40 /r VPMULLD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Multiply the packed dword signed integers in ymm2 and ymm3/m256/m32bcst and store the low 32 bits of each product in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 40 /r VPMULLD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1[1]	Multiply the packed dword signed integers in zmm2 and zmm3/m512/m32bcst and store the low 32 bits of each product in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 40 /r VPMULLQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Multiply the packed qword signed integers in xmm2 and xmm3/m128/m64bcst and store the low 64 bits of each product in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 40 /r VPMULLQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Multiply the packed qword signed integers in ymm2 and ymm3/m256/m64bcst and store the low 64 bits of each product in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 40 /r VPMULLQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ OR AVX10.1[1]	Multiply the packed qword signed integers in zmm2 and zmm3/m512/m64bcst and store the low 64 bits of each product in zmm1 under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a SIMD signed multiply of the packed signed dword/qword integers from each element of the first source operand with the corresponding element in the second source operand. The low 32/64 bits of each 64/128-bit intermediate results are stored to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

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

EVEX encoded versions: 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 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

VPMULLQ (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 Temp[127:0] := SRC1[i+63:i] * SRC2[63:0]
               ELSE Temp[127:0] := SRC1[i+63:i] * SRC2[i+63:i]
           FI;
           DEST[i+63:i] := Temp[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
```

VPMULLD (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 Temp[63:0] := SRC1[i+31:i] * SRC2[31:0]
               ELSE Temp[63:0] := SRC1[i+31:i] * SRC2[i+31:i]
           FI;
           DEST[i+31:i] := Temp[31:0]
       ELSE
           IF *merging-masking*                  ; merging-masking
               *DEST[i+31:i] remains unchanged*
               ELSE                              ; zeroing-masking
                   DEST[i+31:i] := 0
           FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMULLD (VEX.256 Encoded Version)

```
Temp0[63:0] := SRC1[31:0] * SRC2[31:0]
Temp1[63:0] := SRC1[63:32] * SRC2[63:32]
Temp2[63:0] := SRC1[95:64] * SRC2[95:64]
Temp3[63:0] := SRC1[127:96] * SRC2[127:96]
Temp4[63:0] := SRC1[159:128] * SRC2[159:128]
Temp5[63:0] := SRC1[191:160] * SRC2[191:160]
Temp6[63:0] := SRC1[223:192] * SRC2[223:192]
Temp7[63:0] := SRC1[255:224] * SRC2[255:224]

DEST[31:0] := Temp0[31:0]
DEST[63:32] := Temp1[31:0]
DEST[95:64] := Temp2[31:0]
DEST[127:96] := Temp3[31:0]
DEST[159:128] := Temp4[31:0]
DEST[191:160] := Temp5[31:0]
DEST[223:192] := Temp6[31:0]
DEST[255:224] := Temp7[31:0]
DEST[MAXVL-1:256] := 0
```

VPMULLD (VEX.128 Encoded Version)

```
Temp0[63:0] := SRC1[31:0] * SRC2[31:0]
Temp1[63:0] := SRC1[63:32] * SRC2[63:32]
Temp2[63:0] := SRC1[95:64] * SRC2[95:64]
Temp3[63:0] := SRC1[127:96] * SRC2[127:96]
DEST[31:0] := Temp0[31:0]
DEST[63:32] := Temp1[31:0]
DEST[95:64] := Temp2[31:0]
DEST[127:96] := Temp3[31:0]
DEST[MAXVL-1:128] := 0
```

PMULLD (128-bit Legacy SSE Version)

```
Temp0[63:0] := DEST[31:0] * SRC[31:0]
Temp1[63:0] := DEST[63:32] * SRC[63:32]
Temp2[63:0] := DEST[95:64] * SRC[95:64]
Temp3[63:0] := DEST[127:96] * SRC[127:96]
DEST[31:0] := Temp0[31:0]
DEST[63:32] := Temp1[31:0]
DEST[95:64] := Temp2[31:0]
DEST[127:96] := Temp3[31:0]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULLD __m512i _mm512_mullo_epi32(__m512i a, __m512i b);
VPMULLD __m512i _mm512_mask_mullo_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMULLD __m512i _mm512_maskz_mullo_epi32(__mmask16 k, __m512i a, __m512i b);
VPMULLD __m256i _mm256_mask_mullo_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULLD __m256i _mm256_maskz_mullo_epi32(__mmask8 k, __m256i a, __m256i b);
VPMULLD __m128i _mm_mask_mullo_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLD __m128i _mm_maskz_mullo_epi32(__mmask8 k, __m128i a, __m128i b);
VPMULLD __m256i _mm256_mullo_epi32(__m256i a, __m256i b);
PMULLD __m128i _mm_mullo_epi32(__m128i a, __m128i b);
VPMULLQ __m512i _mm512_mullo_epi64(__m512i a, __m512i b);
VPMULLQ __m512i _mm512_mask_mullo_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULLQ __m512i _mm512_maskz_mullo_epi64(__mmask8 k, __m512i a, __m512i b);
VPMULLQ __m256i _mm256_mullo_epi64(__m256i a, __m256i b);
VPMULLQ __m256i _mm256_mask_mullo_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULLQ __m256i _mm256_maskz_mullo_epi64(__mmask8 k, __m256i a, __m256i b);
VPMULLQ __m128i _mm_mullo_epi64(__m128i a, __m128i b);
VPMULLQ __m128i _mm_mask_mullo_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLQ __m128i _mm_maskz_mullo_epi64(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PMULLW—Multiply Packed Signed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D5 /r[1] PMULLW mm, mm/m64	A	V/V	MMX	Multiply the packed signed word integers in mm1 register and mm2/m64, and store the low 16 bits of the results in mm1.
66 0F D5 /r PMULLW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the low 16 bits of the results in xmm1.
VEX.128.66.0F.WIG D5 /r VPMULLW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.
VEX.256.66.0F.WIG D5 /r VPMULLW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1.
EVEX.128.66.0F.WIG D5 /r VPMULLW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the low 16 bits of the results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG D5 /r VPMULLW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG D5 /r VPMULLW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the low 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 1-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: 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. The destination operand is conditionally updated based on writemask k1.

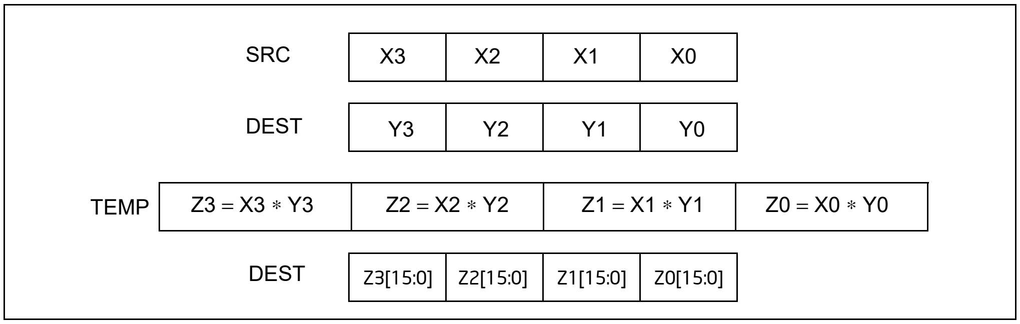


Figure 1-13. PMULLU Instruction Operation Using 64-bit Operands

Operation

PMULLW (With 64-bit Operands)

```
    TEMP0[31:0] := DEST[15:0] ∗ SRC[15:0]; (* Signed multiplication *)
    TEMP1[31:0] := DEST[31:16] ∗ SRC[31:16];
    TEMP2[31:0] := DEST[47:32] ∗ SRC[47:32];
    TEMP3[31:0] := DEST[63:48] ∗ SRC[63:48];
    DEST[15:0] := TEMP0[15:0];
    DEST[31:16] := TEMP1[15:0];
    DEST[47:32] := TEMP2[15:0];
    DEST[63:48] := TEMP3[15:0];
```

PMULLW (With 128-bit Operands)

```
    TEMP0[31:0] := DEST[15:0] ∗ SRC[15:0]; (* Signed multiplication *)
    TEMP1[31:0] := DEST[31:16] ∗ SRC[31:16];
    TEMP2[31:0] := DEST[47:32] ∗ SRC[47:32];
    TEMP3[31:0] := DEST[63:48] ∗ SRC[63:48];
    TEMP4[31:0] := DEST[79:64] ∗ SRC[79:64];
    TEMP5[31:0] := DEST[95:80] ∗ SRC[95:80];
    TEMP6[31:0] := DEST[111:96] ∗ SRC[111:96];
    TEMP7[31:0] := DEST[127:112] ∗ SRC[127:112];
    DEST[15:0] := TEMP0[15:0];
    DEST[31:16] := TEMP1[15:0];
    DEST[47:32] := TEMP2[15:0];
    DEST[63:48] := TEMP3[15:0];
    DEST[79:64] := TEMP4[15:0];
    DEST[95:80] := TEMP5[15:0];
    DEST[111:96] := TEMP6[15:0];
    DEST[127:112] := TEMP7[15:0];
DEST[MAXVL-1:256] := 0
```

VPMULLW (VEX.128 Encoded Version)

```
Temp0[31:0] := SRC1[15:0] * SRC2[15:0]
Temp1[31:0] := SRC1[31:16] * SRC2[31:16]
Temp2[31:0] := SRC1[47:32] * SRC2[47:32]
Temp3[31:0] := SRC1[63:48] * SRC2[63:48]
Temp4[31:0] := SRC1[79:64] * SRC2[79:64]
Temp5[31:0] := SRC1[95:80] * SRC2[95:80]
Temp6[31:0] := SRC1[111:96] * SRC2[111:96]
Temp7[31:0] := SRC1[127:112] * SRC2[127:112]
DEST[15:0] := Temp0[15:0]
DEST[31:16] := Temp1[15:0]
DEST[47:32] := Temp2[15:0]
DEST[63:48] := Temp3[15:0]
DEST[79:64] := Temp4[15:0]
DEST[95:80] := Temp5[15:0]
DEST[111:96] := Temp6[15:0]
DEST[127:112] := Temp7[15:0]
DEST[MAXVL-1:128] := 0
```

PMULLW (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
            temp[31:0] := SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] := temp[15:0]
        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] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULLW __m512i _mm512_mullo_epi16(__m512i a, __m512i b);
VPMULLW __m512i _mm512_mask_mullo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULLW __m512i _mm512_maskz_mullo_epi16( __mmask32 k, __m512i a, __m512i b);
VPMULLW __m256i _mm256_mask_mullo_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULLW __m256i _mm256_maskz_mullo_epi16( __mmask16 k, __m256i a, __m256i b);
VPMULLW __m128i _mm_mask_mullo_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLW __m128i _mm_maskz_mullo_epi16( __mmask8 k, __m128i a, __m128i b);
PMULLW __m64 _mm_mullo_pi16(__m64 m1, __m64 m2)
(V)PMULLW __m128i _mm_mullo_epi16 ( __m128i a, __m128i b)
VPMULLW __m256i _mm256_mullo_epi16 ( __m256i a, __m256i b);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PMULUDQ—Multiply Packed Unsigned Doubleword Integers

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F F4 /r[1] PMULUDQ mm1, mm2/m64	A	V/V	SSE2	Multiply unsigned doubleword integer in mm1 by unsigned doubleword integer in mm2/m64, and store the quadword result in mm1.
66 0F F4 /r PMULUDQ xmm1, xmm2/m128	A	V/V	SSE2	Multiply packed unsigned doubleword integers in xmm1 by packed unsigned doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.128.66.0F.WIG F4 /r VPMULUDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.256.66.0F.WIG F4 /r VPMULUDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256, and store the quadword results in ymm1.
EVEX.128.66.0F.W1 F4 /r VPMULUDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 under writemask k1.
EVEX.256.66.0F.W1 F4 /r VPMULUDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 under writemask k1.
EVEX.512.66.0F.W1 F4 /r VPMULUDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1[2]	Multiply packed unsigned doubleword integers in zmm2 by packed unsigned doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register or a 64-bit memory location. The destination operand can be an unsigned doubleword integer stored in the low doubleword an MMX technology register. The result is an unsigned quadword integer stored in the destination an MMX technology register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation.

128-bit Legacy SSE version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is four packed unsigned doubleword integers stored in the first (low), third, fifth, and seventh doublewords of a YMM register or a 256-bit memory location. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth, and seventh doublewords are used in the computation. The first source operand is four packed unsigned doubleword integers stored in the first, third, fifth, and seventh doublewords of an YMM register. The destination contains four packed unaligned quadword integers stored in an YMM register.

EVEX encoded version: The input unsigned doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an 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 is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

Operation

PMULUDQ (With 64-Bit Operands)

```
    DEST[63:0] := DEST[31:0] ∗ SRC[31:0];
```

PMULUDQ (With 128-Bit Operands)

```
    DEST[63:0] := DEST[31:0] ∗ SRC[31:0];
    DEST[127:64] := DEST[95:64] ∗ SRC[95:64];
```

VPMULUDQ (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[31:0] * SRC2[31:0]
DEST[127:64] := SRC1[95:64] * SRC2[95:64]
DEST[MAXVL-1:128] := 0
```

VPMULUDQ (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[31:0] * SRC2[31:0]
DEST[127:64] := SRC1[95:64] * SRC2[95:64
DEST[191:128] := SRC1[159:128] * SRC2[159:128]
DEST[255:192] := SRC1[223:192] * SRC2[223:192]
DEST[MAXVL-1:256] := 0
```

VPMULUDQ (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] := ZeroExtend64( SRC1[i+31:i]) * ZeroExtend64( SRC2[31:0] )
               ELSE DEST[i+63:i] := ZeroExtend64( SRC1[i+31:i]) * ZeroExtend64( SRC2[i+31: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

```
VPMULUDQ __m512i _mm512_mul_epu32(__m512i a, __m512i b);
VPMULUDQ __m512i _mm512_mask_mul_epu32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m512i _mm512_maskz_mul_epu32( __mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m256i _mm256_mask_mul_epu32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m256i _mm256_maskz_mul_epu32( __mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m128i _mm_mask_mul_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULUDQ __m128i _mm_maskz_mul_epu32( __mmask8 k, __m128i a, __m128i b);
PMULUDQ __m64 _mm_mul_su32 (__m64 a, __m64 b)
(V)PMULUDQ __m128i _mm_mul_epu32 ( __m128i a, __m128i b)
VPMULUDQ __m256i _mm256_mul_epu32( __m256i a, __m256i b);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

POR—Bitwise Logical OR

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EB /r[1] POR mm, mm/m64	A	V/V	MMX	Bitwise OR of mm/m64 and mm.
66 0F EB /r POR xmm1, xmm2/m128	A	V/V	SSE2	Bitwise OR of xmm2/m128 and xmm1.
VEX.128.66.0F.WIG EB /r VPOR xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise OR of xmm2/m128 and xmm3.
VEX.256.66.0F.WIG EB /r VPOR ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Bitwise OR of ymm2/m256 and ymm3.
EVEX.128.66.0F.W0 EB /r VPORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise OR of packed doubleword integers in xmm2 and xmm3/m128/m32bcst using writemask k1.
EVEX.256.66.0F.W0 EB /r VPORD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise OR of packed doubleword integers in ymm2 and ymm3/m256/m32bcst using writemask k1.
EVEX.512.66.0F.W0 EB /r VPORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1[2]	Bitwise OR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.128.66.0F.W1 EB /r VPORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise OR of packed quadword integers in xmm2 and xmm3/m128/m64bcst using writemask k1.
EVEX.256.66.0F.W1 EB /r VPORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise OR of packed quadword integers in ymm2 and ymm3/m256/m64bcst using writemask k1.
EVEX.512.66.0F.W1 EB /r VPORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1[2]	Bitwise OR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a bitwise logical OR operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source and destination operands can be YMM registers.

EVEX encoded version: 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/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

Operation

POR (64-bit Operand)

```
DEST := DEST OR SRC
```

POR (128-bit Legacy SSE Version)

```
DEST := DEST OR SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPOR (VEX.128 Encoded Version)

```
DEST := SRC1 OR SRC2
DEST[MAXVL-1:128] := 0
```

VPOR (VEX.256 Encoded Version)

```
DEST := SRC1 OR SRC2
DEST[MAXVL-1:256] := 0
```

VPORD (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 OR SRC2[31:0]
               ELSE DEST[i+31:i] := SRC1[i+31:i] BITWISE OR SRC2[i+31:i]
            FI;
         ELSE
            IF *merging-masking*                    ; merging-masking
               *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

```
VPORD __m512i _mm512_or_epi32(__m512i a, __m512i b);
VPORD __m512i _mm512_mask_or_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPORD __m512i _mm512_maskz_or_epi32( __mmask16 k, __m512i a, __m512i b);
VPORD __m256i _mm256_or_epi32(__m256i a, __m256i b);
VPORD __m256i _mm256_mask_or_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b,);
VPORD __m256i _mm256_maskz_or_epi32( __mmask8 k, __m256i a, __m256i b);
VPORD __m128i _mm_or_epi32(__m128i a, __m128i b);
VPORD __m128i _mm_mask_or_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPORD __m128i _mm_maskz_or_epi32( __mmask8 k, __m128i a, __m128i b);
VPORQ __m512i _mm512_or_epi64(__m512i a, __m512i b);
VPORQ __m512i _mm512_mask_or_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPORQ __m512i _mm512_maskz_or_epi64(__mmask8 k, __m512i a, __m512i b);
VPORQ __m256i _mm256_or_epi64(__m256i a, int imm);
VPORQ __m256i _mm256_mask_or_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPORQ __m256i _mm256_maskz_or_epi64( __mmask8 k, __m256i a, __m256i b);
VPORQ __m128i _mm_or_epi64(__m128i a, __m128i b);
VPORQ __m128i _mm_mask_or_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPORQ __m128i _mm_maskz_or_epi64( __mmask8 k, __m128i a, __m128i b);
POR __m64 _mm_or_si64(__m64 m1, __m64 m2)
(V)POR __m128i _mm_or_si128(__m128i m1, __m128i m2)
VPOR __m256i _mm256_or_si256 ( __m256i a, __m256i b)
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PSADBW—Compute Sum of Absolute Differences

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F6 /r[1] PSADBW mm1, mm2/m64	A	V/V	SSE	Computes the absolute differences of the packed unsigned byte integers from mm2 /m64 and mm1; differences are then summed to produce an unsigned word integer result.
66 0F F6 /r PSADBW xmm1, xmm2/m128	A	V/V	SSE2	Computes the absolute differences of the packed unsigned byte integers from xmm2 /m128 and xmm1; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Computes the absolute differences of the packed unsigned byte integers from xmm3 /m128 and xmm2; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Computes the absolute differences of the packed unsigned byte integers from ymm3 /m256 and ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Computes the absolute differences of the packed unsigned byte integers from xmm3 /m128 and xmm2; then each consecutive 8 differences are summed separately to produce two unsigned word integer results.
EVEX.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Computes the absolute differences of the packed unsigned byte integers from ymm3 /m256 and ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.512.66.0F.WIG F6 /r VPSADBW zmm1, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Computes the absolute differences of the packed unsigned byte integers from zmm3 /m512 and zmm2; then each consecutive 8 differences are summed separately to produce eight unsigned word integer results.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	N/A

Description

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (second operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. Figure 1-14 shows the operation of the PSADBW instruction when using 64-bit operands.

When operating on 64-bit operands, the word integer result is stored in the low word of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

When operating on 128-bit operands, two packed results are computed. Here, the 8 low-order bytes of the source and destination operands are operated on to produce a word result that is stored in the low word of the destination operand, and the 8 high-order bytes are operated on to produce a word result that is stored in bits 64 through 79 of the destination operand. The remaining bytes of the destination operand are cleared.

For 256-bit version, the third group of 8 differences are summed to produce an unsigned word in bits[143:128] of the destination register and the fourth group of 8 differences are summed to produce an unsigned word in bits[207:192] of the destination register. The remaining words of the destination are set to 0.

For 512-bit version, the fifth group result is stored in bits [271:256] of the destination. The result from the sixth group is stored in bits [335:320]. The results for the seventh and eighth group are stored respectively in bits [399:384] and bits [463:447], respectively. The remaining bits in the destination are set to 0.

In 64-bit mode and not encoded by VEX/EVEX prefix, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 and EVEX.256 encoded versions: The first source operand and destination register are YMM registers. The second source operand is an YMM register or a 256-bit memory location. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The first source operand and destination register are ZMM registers. The second source operand is a ZMM register or a 512-bit memory location.

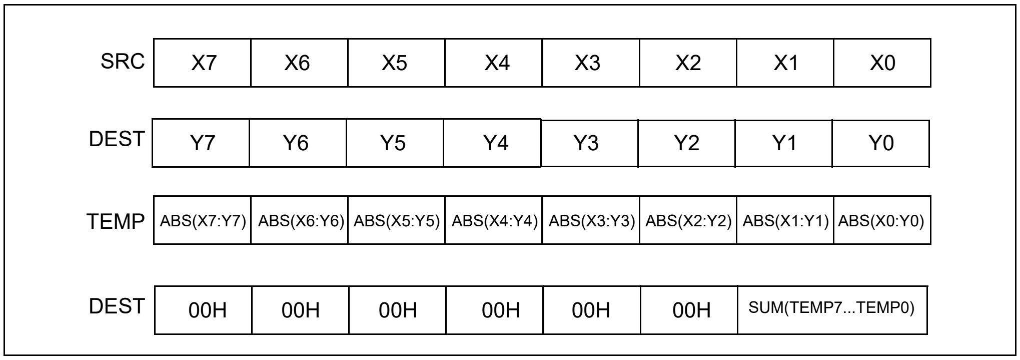


Figure 1-14. PSADBW Instruction Operation Using 64-bit Operands

Operation

VPSADBW (EVEX Encoded Versions)

```
VL = 128, 256, 512
TEMP0 := ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 1 through 15 *)
TEMP15 := ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 00000000000H

IF VL >= 256
    (* Repeat operation for bytes 16 through 31*)
    TEMP31 := ABS(SRC1[255:248] - SRC2[255:248])
    DEST[143:128] := SUM(TEMP16:TEMP23)
    DEST[191:144] := 000000000000H
    DEST[207:192] := SUM(TEMP24:TEMP31)
    DEST[223:208] := 00000000000H
FI;
IF VL >= 512
(* Repeat operation for bytes 32 through 63*)
    TEMP63 := ABS(SRC1[511:504] - SRC2[511:504])
    DEST[271:256] := SUM(TEMP0:TEMP7)
    DEST[319:272] := 000000000000H
    DEST[335:320] := SUM(TEMP8:TEMP15)
    DEST[383:336] := 00000000000H
    DEST[399:384] := SUM(TEMP16:TEMP23)
    DEST[447:400] := 000000000000H
    DEST[463:448] := SUM(TEMP24:TEMP31)
    DEST[511:464] := 00000000000H
FI;
DEST[MAXVL-1:VL] := 0
```

VPSADBW (VEX.256 Encoded Version)

```
TEMP0 := ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 30*)
TEMP31 := ABS(SRC1[255:248] - SRC2[255:248])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 00000000000H
DEST[143:128] := SUM(TEMP16:TEMP23)
DEST[191:144] := 000000000000H
DEST[207:192] := SUM(TEMP24:TEMP31)
DEST[223:208] := 00000000000H
DEST[MAXVL-1:256] := 0
```

VPSADBW (VEX.128 Encoded Version)

```
TEMP0 := ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 := ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 00000000000H
DEST[MAXVL-1:128] := 0
```

PSADBW (128-bit Legacy SSE Version)

```
TEMP0 := ABS(DEST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 := ABS(DEST[127:120] - SRC[127:120])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 00000000000
DEST[MAXVL-1:128] (Unmodified)
```

PSADBW (64-bit Operand)

```
TEMP0 := ABS(DEST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 6 *)
TEMP7 := ABS(DEST[63:56] - SRC[63:56])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSADBW __m512i _mm512_sad_epu8(__m512i a, __m512i b)
PSADBW __m64 _mm_sad_pu8(__m64 a,__m64 b)
(V)PSADBW __m128i _mm_sad_epu8(__m128i a, __m128i b)
VPSADBW __m256i _mm256_sad_epu8(__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

PSHUFB—Packed Shuffle Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 00 /r[1] PSHUFB mm1, mm2/m64	A	V/V	SSSE3	Shuffle bytes in mm1 according to contents of mm2/m64.
66 0F 38 00 /r PSHUFB xmm1, xmm2/m128	A	V/V	SSSE3	Shuffle bytes in xmm1 according to contents of xmm2/m128.
VEX.128.66.0F38.WIG 00 /r VPSHUFB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Shuffle bytes in xmm2 according to contents of xmm3/m128.
VEX.256.66.0F38.WIG 00 /r VPSHUFB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Shuffle bytes in ymm2 according to contents of ymm3/m256.
EVEX.128.66.0F38.WIG 00 /r VPSHUFB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Shuffle bytes in xmm2 according to contents of xmm3/m128 under write mask k1.
EVEX.256.66.0F38.WIG 00 /r VPSHUFB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Shuffle bytes in ymm2 according to contents of ymm3/m256 under write mask k1.
EVEX.512.66.0F38.WIG 00 /r VPSHUFB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Shuffle bytes in zmm2 according to contents of zmm3/m512 under write mask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

PSHUFB performs in-place shuffles of bytes in the destination operand (the first operand) according to the shuffle control mask in the source operand (the second operand). The instruction permutes the data in the destination operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte in the shuffle control mask forms an index to permute the corresponding byte in the destination operand. The value of each index is the least significant 4 bits (128-bit operation) or 3 bits (64-bit operation) of the shuffle control byte. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is the first operand, the first source operand is the second operand, the second source operand is the third operand. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Bits (255:128) of the destination YMM register stores the 16-byte shuffle result of the upper 16 bytes of the first source operand, using the upper 16-bytes of the second source operand as control mask. The value of each index is for the high 128-bit lane is the least significant 4 bits of the respective shuffle control byte. The index value selects a source data element within each 128-bit lane.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX and VEX encoded version: Four/two in-lane 128-bit shuffles.

Operation

PSHUFB (With 64-bit Operands)

```
TEMP := DEST
for i = 0 to 7 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7...(i*8)+0] := 0;
    else
        index[2..0] := SRC[(i*8)+2 .. (i*8)+0];
        DEST[(i*8)+7...(i*8)+0] := TEMP[(index*8+7)..(index*8+0)];
    endif;
}
PSHUFB (with 128 bit operands)
TEMP := DEST
for i = 0 to 15 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7..(i*8)+0] := 0;
     else
        index[3..0] := SRC[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] := TEMP[(index*8+7)..(index*8+0)];
    endif
}
```

VPSHUFB (VEX.128 Encoded Version)

```
for i = 0 to 15 {
    if (SRC2[(i * 8)+7] = 1) then
        DEST[(i*8)+7..(i*8)+0] := 0;
        else
        index[3..0] := SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] := SRC1[(index*8+7)..(index*8+0)];
    endif
}
DEST[MAXVL-1:128] := 0
```

VPSHUFB (VEX.256 Encoded Version)

```
for i = 0 to 15 {
    if (SRC2[(i * 8)+7] == 1 ) then
        DEST[(i*8)+7..(i*8)+0] := 0;
        else
        index[3..0] := SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] := SRC1[(index*8+7)..(index*8+0)];
    endif
    if (SRC2[128 + (i * 8)+7] == 1 ) then
```

```
            DEST[128 + (i*8)+7..(i*8)+0] := 0;
            else
            index[3..0] := SRC2[128 + (i*8)+3 .. (i*8)+0];
            DEST[128 + (i*8)+7..(i*8)+0] := SRC1[128 + (index*8+7)..(index*8+0)];
        endif
}
```

VPSHUFB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
jmask := (KL-1) & ~0xF                          // 0x00, 0x10, 0x30 depending on the VL
FOR j = 0 TO KL-1                               // dest
    IF kl[ i ] or no_masking
        index := src.byte[ j ];
        IF index & 0x80
            Dest.byte[ j ] := 0;
        ELSE
            index := (index & 0xF) + (j & jmask);   // 16-element in-lane lookup
            Dest.byte[ j ] := src.byte[ index ];
    ELSE if zeroing
        Dest.byte[ j ] := 0;
DEST[MAXVL-1:VL] := 0;
```

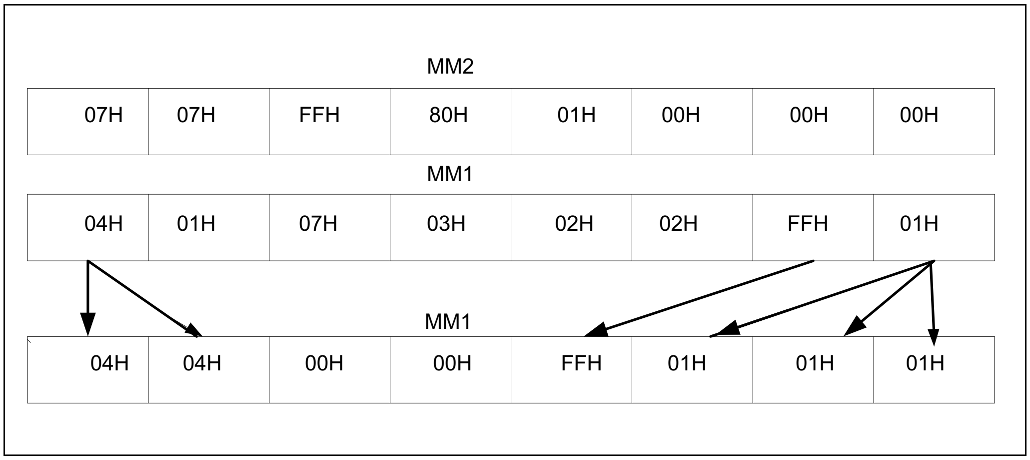

MM2
07H 07H FFH 80H 01H 00H 00H 00H
MM1
04H 01H 07H 03H 02H 02H FFH 01H
MM1
04H 04H 00H 00H FFH 01H 01H 01H

Figure 1-15. PSHUFB with 64-Bit Operands

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFB __m512i _mm512_shuffle_epi8(__m512i a, __m512i b);
VPSHUFB __m512i _mm512_mask_shuffle_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSHUFB __m512i _mm512_maskz_shuffle_epi8(__mmask64 k, __m512i a, __m512i b);
VPSHUFB __m256i _mm256_mask_shuffle_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSHUFB __m256i _mm256_maskz_shuffle_epi8(__mmask32 k, __m256i a, __m256i b);
VPSHUFB __m128i _mm_mask_shuffle_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSHUFB __m128i _mm_maskz_shuffle_epi8(__mmask16 k, __m128i a, __m128i b);
PSHUFB: __m64 _mm_shuffle_pi8 (__m64 a, __m64 b)
(V)PSHUFB: __m128i _mm_shuffle_epi8 (__m128i a, __m128i b)
VPSHUFB:__m256i _mm256_shuffle_epi8(__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

PSHUFD—Shuffle Packed Doublewords

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 70 /r ib PSHUFD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.128.66.0F.WIG 70 /r ib VPSHUFD xmm1, xmm2/m128, imm8	A	V/V	AVX	Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.256.66.0F.WIG 70 /r ib VPSHUFD ymm1, ymm2/m256, imm8	A	V/V	AVX2	Shuffle the doublewords in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.
EVEX.128.66.0F.W0 70 /r ib VPSHUFD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Shuffle the doublewords in xmm2/m128/m32bcst based on the encoding in imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F.W0 70 /r ib VPSHUFD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Shuffle the doublewords in ymm2/m256/m32bcst based on the encoding in imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F.W0 70 /r ib VPSHUFD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	B	V/V	AVX512F OR AVX10.1[1]	Shuffle the doublewords in zmm2/m512/m32bcst based on the encoding in imm8 and store the result in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at the locations selected with the order operand (third operand). Figure 1-16 shows the operation of the 256-bit VPSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location within a 128-bit lane and copy to the target element in the destination operand. For example, bits 0 and 1 of the order operand targets the first doubleword element in the low and high 128-bit lane of the destination operand for 256-bit VPSHUFD. The encoded value of bits 1:0 of the order operand (see the field encoding in Figure 1-16) determines which doubleword element (from the respective 128-bit lane) of the source operand will be copied to doubleword 0 of the destination operand.

For 128-bit operation, only the low 128-bit lane are operative. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

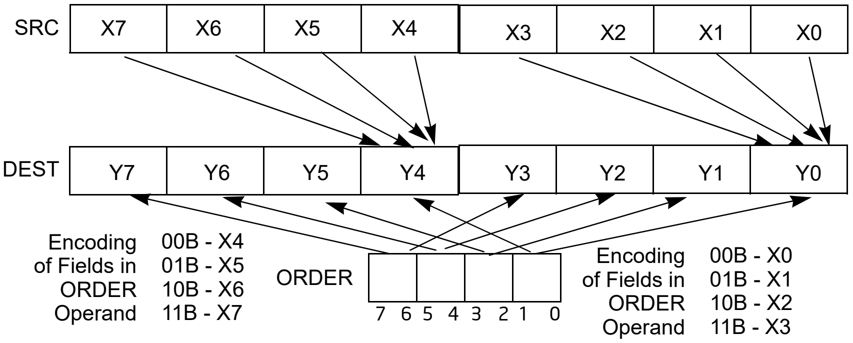


Figure 1-16. 256-bit VPSHUFD Instruction Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

In 64-bit mode and not encoded in VEX/EVEX, using REX.R permits this instruction to access XMM8-XMM15.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

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

VEX.256 encoded version: The source operand can be an YMM register or a 256-bit memory location. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. Bits (255-1:128) of the destination stores the shuffled results of the upper 16 bytes of the source operand using the immediate byte as the order operand.

EVEX encoded version: The source operand can be an 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 updated according to the writemask.

Each 128-bit lane of the destination stores the shuffled results of the respective lane of the source operand using the immediate byte as the order operand.

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

Operation

PSHUFD (128-bit Legacy SSE Version)

```
DEST[31:0] := (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] := (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] := (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] := (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:128] (Unmodified)
```

VPSHUFD (VEX.128 Encoded Version)

```
DEST[31:0] := (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] := (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] := (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] := (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:128] := 0
```

VPSHUFD (VEX.256 Encoded Version)

```
DEST[31:0] := (SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] := (SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] := (SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] := (SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
DEST[159:128] := (SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
DEST[191:160] := (SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
DEST[223:192] := (SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
DEST[255:224] := (SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:256] := 0
```

VPSHUFD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN TMP_SRC[i+31:i] := SRC[31:0]
        ELSE TMP_SRC[i+31:i] := SRC[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[31:0] := (TMP_SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[63:32] := (TMP_SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[95:64] := (TMP_SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[127:96] := (TMP_SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
FI;
IF VL >= 256
    TMP_DEST[159:128] := (TMP_SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[191:160] := (TMP_SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[223:192] := (TMP_SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[255:224] := (TMP_SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
FI;
IF VL >= 512
    TMP_DEST[287:256] := (TMP_SRC[383:256] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[319:288] := (TMP_SRC[383:256] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[351:320] := (TMP_SRC[383:256] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[383:352] := (TMP_SRC[383:256] >> (ORDER[7:6] * 32))[31:0];
    TMP_DEST[415:384] := (TMP_SRC[511:384] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[447:416] := (TMP_SRC[511:384] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[479:448] := (TMP_SRC[511:384] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[511:480] := (TMP_SRC[511:384] >> (ORDER[7:6] * 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*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
```

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

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFD __m512i _mm512_shuffle_epi32(__m512i a, int n);
VPSHUFD __m512i _mm512_mask_shuffle_epi32(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFD __m512i _mm512_maskz_shuffle_epi32(__mmask16 k, __m512i a, int n);
VPSHUFD __m256i _mm256_mask_shuffle_epi32(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFD __m256i _mm256_maskz_shuffle_epi32(__mmask8 k, __m256i a, int n);
VPSHUFD __m128i _mm_mask_shuffle_epi32(__m128i s, __mmask8 k, __m128i a, int n);
VPSHUFD __m128i _mm_maskz_shuffle_epi32(__mmask8 k, __m128i a, int n);
(V)PSHUFD __m128i _mm_shuffle_epi32(__m128i a, int n)
VPSHUFD __m256i _mm256_shuffle_epi32(__m256i a, const int n)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”

Additionally:

#UD If VEX.vvvv ≠ 1111B or EVEX.vvvv ≠ 1111B.

PSHUFHW—Shuffle Packed High Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 70 /r ib PSHUFHW xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.128.F3.0F.WIG 70 /r ib VPSHUFHW xmm1, xmm2/m128, imm8	A	V/V	AVX	Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.256.F3.0F.WIG 70 /r ib VPSHUFHW ymm1, ymm2/m256, imm8	A	V/V	AVX2	Shuffle the high words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.
EVEX.128.F3.0F.WIG 70 /r ib VPSHUFHW xmm1 {k1}{z}, xmm2/m128, imm8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1 under write mask k1.
EVEX.256.F3.0F.WIG 70 /r ib VPSHUFHW ymm1 {k1}{z}, ymm2/m256, imm8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Shuffle the high words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1 under write mask k1.
EVEX.512.F3.0F.WIG 70 /r ib VPSHUFHW zmm1 {k1}{z}, zmm2/m512, imm8	B	V/V	AVX512BW OR AVX10.1[1]	Shuffle the high words in zmm2/m512 based on the encoding in imm8 and store the result in zmm1 under write mask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Copies words from the high quadword of a 128-bit lane of the source operand and inserts them in the high quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. This 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 1-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3, 4) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the source operand is copied to the low quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the write-mask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

PSHUFHW (128-bit Legacy SSE Version)

```
DEST[63:0] := SRC[63:0]
DEST[79:64] := (SRC >> (imm[1:0] *16))[79:64]
DEST[95:80] := (SRC >> (imm[3:2] * 16))[79:64]
DEST[111:96] := (SRC >> (imm[5:4] * 16))[79:64]
DEST[127:112] := (SRC >> (imm[7:6] * 16))[79:64]
DEST[MAXVL-1:128] (Unmodified)
```

VPSHUFHW (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0]
DEST[79:64] := (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] := (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] := (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] := (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[MAXVL-1:128] := 0
```

VPSHUFHW (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0]
DEST[79:64] := (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] := (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] := (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] := (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[191:128] := SRC1[191:128]
DEST[207192] := (SRC1 >> (imm[1:0] *16))[207:192]
DEST[223:208] := (SRC1 >> (imm[3:2] * 16))[207:192]
DEST[239:224] := (SRC1 >> (imm[5:4] * 16))[207:192]
DEST[255:240] := (SRC1 >> (imm[7:6] * 16))[207:192]
DEST[MAXVL-1:256] := 0
```

VPSHUFHW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL >= 128
    TMP_DEST[63:0] := SRC1[63:0]
    TMP_DEST[79:64] := (SRC1 >> (imm[1:0] *16))[79:64]
    TMP_DEST[95:80] := (SRC1 >> (imm[3:2] * 16))[79:64]
    TMP_DEST[111:96] := (SRC1 >> (imm[5:4] * 16))[79:64]
    TMP_DEST[127:112] := (SRC1 >> (imm[7:6] * 16))[79:64]
FI;
IF VL >= 256
    TMP_DEST[191:128] := SRC1[191:128]
    TMP_DEST[207:192] := (SRC1 >> (imm[1:0] *16))[207:192]
    TMP_DEST[223:208] := (SRC1 >> (imm[3:2] * 16))[207:192]
```

```
    TMP_DEST[239:224] := (SRC1 >> (imm[5:4] * 16))[207:192]
    TMP_DEST[255:240] := (SRC1 >> (imm[7:6] * 16))[207:192]
FI;
IF VL >= 512
    TMP_DEST[319:256] := SRC1[319:256]
    TMP_DEST[335:320] := (SRC1 >> (imm[1:0] *16))[335:320]
    TMP_DEST[351:336] := (SRC1 >> (imm[3:2] * 16))[335:320]
    TMP_DEST[367:352] := (SRC1 >> (imm[5:4] * 16))[335:320]
    TMP_DEST[383:368] := (SRC1 >> (imm[7:6] * 16))[335:320]
    TMP_DEST[447:384] := SRC1[447:384]
    TMP_DEST[463:448] := (SRC1 >> (imm[1:0] *16))[463:448]
    TMP_DEST[479:464] := (SRC1 >> (imm[3:2] * 16))[463:448]
    TMP_DEST[495:480] := (SRC1 >> (imm[5:4] * 16))[463:448]
    TMP_DEST[511:496] := (SRC1 >> (imm[7:6] * 16))[463:448]
FI;

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*                   ; zeroing-masking
                        DEST[i+15:i] := 0
              FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFHW __m512i _mm512_shufflehi_epi16(__m512i a, int n);
VPSHUFHW __m512i _mm512_mask_shufflehi_epi16(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFHW __m512i _mm512_maskz_shufflehi_epi16(__mmask16 k, __m512i a, int n);
VPSHUFHW __m256i _mm256_mask_shufflehi_epi16(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFHW __m256i _mm256_maskz_shufflehi_epi16(__mmask8 k, __m256i a, int n);
VPSHUFHW __m128i _mm_mask_shufflehi_epi16(__m128i s, __mmask8 k, __m128i a, int n);
VPSHUFHW __m128i _mm_maskz_shufflehi_epi16(__mmask8 k, __m128i a, int n);
(V)PSHUFHW __m128i _mm_shufflehi_epi16(__m128i a, int n)
VPSHUFHW __m256i _mm256_shufflehi_epi16(__m256i a, const int n)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

Additionally:

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

PSHUFLW—Shuffle Packed Low Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 70 /r ib PSHUFLW xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.128.F2.0F.WIG 70 /r ib VPSHUFLW xmm1, xmm2/m128, imm8	A	V/V	AVX	Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.256.F2.0F.WIG 70 /r ib VPSHUFLW ymm1, ymm2/m256, imm8	A	V/V	AVX2	Shuffle the low words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.
EVEX.128.F2.0F.WIG 70 /r ib VPSHUFLW xmm1 {k1}{z}, xmm2/m128, imm8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1 under write mask k1.
EVEX.256.F2.0F.WIG 70 /r ib VPSHUFLW ymm1 {k1}{z}, ymm2/m256, imm8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Shuffle the low words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1 under write mask k1.
EVEX.512.F2.0F.WIG 70 /r ib VPSHUFLW zmm1 {k1}{z}, zmm2/m512, imm8	B	V/V	AVX512BW OR AVX10.1[1]	Shuffle the low words in zmm2/m512 based on the encoding in imm8 and store the result in zmm1 under write mask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Copies words from the low quadword of a 128-bit lane of the source operand and inserts them in the low quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. The 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 1-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3) from the low quadword of the source operand to be copied to the destination operand. The high quadword of the source operand is copied to the high quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

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

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

PSHUFLW (128-bit Legacy SSE Version)

```
DEST[15:0] := (SRC >> (imm[1:0] *16))[15:0]
DEST[31:16] := (SRC >> (imm[3:2] * 16))[15:0]
DEST[47:32] := (SRC >> (imm[5:4] * 16))[15:0]
DEST[63:48] := (SRC >> (imm[7:6] * 16))[15:0]
DEST[127:64] := SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

VPSHUFLW (VEX.128 Encoded Version)

```
DEST[15:0] := (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] := (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] := (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] := (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] := SRC[127:64]
DEST[MAXVL-1:128] := 0
```

VPSHUFLW (VEX.256 Encoded Version)

```
DEST[15:0] := (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] := (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] := (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] := (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] := SRC1[127:64]
DEST[143:128] := (SRC1 >> (imm[1:0] *16))[143:128]
DEST[159:144] := (SRC1 >> (imm[3:2] * 16))[143:128]
DEST[175:160] := (SRC1 >> (imm[5:4] * 16))[143:128]
DEST[191:176] := (SRC1 >> (imm[7:6] * 16))[143:128]
DEST[255:192] := SRC1[255:192]
DEST[MAXVL-1:256] := 0
```

VPSHUFLW (EVEX.U1.512 Encoded Version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL >= 128
    TMP_DEST[15:0] := (SRC1 >> (imm[1:0] *16))[15:0]
    TMP_DEST[31:16] := (SRC1 >> (imm[3:2] * 16))[15:0]
    TMP_DEST[47:32] := (SRC1 >> (imm[5:4] * 16))[15:0]
    TMP_DEST[63:48] := (SRC1 >> (imm[7:6] * 16))[15:0]
    TMP_DEST[127:64] := SRC1[127:64]
FI;
IF VL >= 256
    TMP_DEST[143:128] := (SRC1 >> (imm[1:0] *16))[143:128]
    TMP_DEST[159:144] := (SRC1 >> (imm[3:2] * 16))[143:128]
    TMP_DEST[175:160] := (SRC1 >> (imm[5:4] * 16))[143:128]
    TMP_DEST[191:176] := (SRC1 >> (imm[7:6] * 16))[143:128]
    TMP_DEST[255:192] := SRC1[255:192]
FI;
IF VL >= 512
```

```
    TMP_DEST[271:256] := (SRC1 >> (imm[1:0] *16))[271:256]
    TMP_DEST[287:272] := (SRC1 >> (imm[3:2] * 16))[271:256]
    TMP_DEST[303:288] := (SRC1 >> (imm[5:4] * 16))[271:256]
    TMP_DEST[319:304] := (SRC1 >> (imm[7:6] * 16))[271:256]
    TMP_DEST[383:320] := SRC1[383:320]
    TMP_DEST[399:384] := (SRC1 >> (imm[1:0] *16))[399:384]
    TMP_DEST[415:400] := (SRC1 >> (imm[3:2] * 16))[399:384]
    TMP_DEST[431:416] := (SRC1 >> (imm[5:4] * 16))[399:384]
    TMP_DEST[447:432] := (SRC1 >> (imm[7:6] * 16))[399:384]
    TMP_DEST[511:448] := SRC1[511:448]
FI;

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*          ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSHUFLW __m512i _mm512_shufflelo_epi16(__m512i a, int n);
VPSHUFLW __m512i _mm512_mask_shufflelo_epi16(__m512i s, __mmask16 k, __m512i a, int n );
VPSHUFLW __m512i _mm512_maskz_shufflelo_epi16( __mmask16 k, __m512i a, int n );
VPSHUFLW __m256i _mm256_mask_shufflelo_epi16(__m256i s, __mmask8 k, __m256i a, int n );
VPSHUFLW __m256i _mm256_maskz_shufflelo_epi16( __mmask8 k, __m256i a, int n );
VPSHUFLW __m128i _mm_mask_shufflelo_epi16(__m128i s, __mmask8 k, __m128i a, int n );
VPSHUFLW __m128i _mm_maskz_shufflelo_epi16( __mmask8 k, __m128i a, int n );
(V)PSHUFLW:__m128i _mm_shufflelo_epi16(__m128i a, int n)
VPSHUFLW:__m256i _mm256_shufflelo_epi16(__m256i a, const int n)
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

Additionally:

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

PSLLDQ—Shift Double Quadword Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /7 ib PSLLDQ xmm1, imm8	A	V/V	SSE2	Shift xmm1 left by imm8 bytes while shifting in 0s.
VEX.128.66.0F.WIG 73 /7 ib VPSLLDQ xmm1, xmm2, imm8	B	V/V	AVX	Shift xmm2 left by imm8 bytes while shifting in 0s and store result in xmm1.
VEX.256.66.0F.WIG 73 /7 ib VPSLLDQ ymm1, ymm2, imm8	B	V/V	AVX2	Shift ymm2 left by imm8 bytes while shifting in 0s and store result in ymm1.
EVEX.128.66.0F.WIG 73 /7 ib VPSLLDQ xmm1,xmm2/ m128, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Shift xmm2/m128 left by imm8 bytes while shifting in 0s and store result in xmm1.
EVEX.256.66.0F.WIG 73 /7 ib VPSLLDQ ymm1, ymm2/m256, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Shift ymm2/m256 left by imm8 bytes while shifting in 0s and store result in ymm1.
EVEX.512.66.0F.WIG 73 /7 ib VPSLLDQ zmm1, zmm2/m512, imm8	C	V/V	AVX512BW OR AVX10.1[1]	Shift zmm2/m512 left by imm8 bytes while shifting in 0s and store result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (r, w)	imm8	N/A	N/A
B	N/A	VEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
C	Full Mem	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A

Description

Shifts the destination operand (first operand) to the left by the number of bytes specified in the count operand (second operand). The empty low-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

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. The count operand applies to each 128-bit lanes.

Operation

VPSLLDQ (EVEX.U1.512 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] << (TEMP * 8)
DEST[255:128] := SRC[255:128] << (TEMP * 8)
DEST[383:256] := SRC[383:256] << (TEMP * 8)
DEST[511:384] := SRC[511:384] << (TEMP * 8)
DEST[MAXVL-1:512] := 0
```

VPSLLDQ (VEX.256 and EVEX.256 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] << (TEMP * 8)
DEST[255:128] := SRC[255:128] << (TEMP * 8)
DEST[MAXVL-1:256] := 0
```

VPSLLDQ (VEX.128 and EVEX.128 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := SRC << (TEMP * 8)
DEST[MAXVL-1:128] := 0
```

PSLLDQ(128-bit Legacy SSE Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := DEST << (TEMP * 8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)PSLLDQ __m128i _mm_slli_si128 ( __m128i a, int imm)
VPSLLDQ __m256i _mm256_slli_si256 ( __m256i a, const int imm)
VPSLLDQ __m512i _mm512_bslli_epi128 ( __m512i a, const int imm)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F1 /r[1] PSLLW mm, mm/m64	A	V/V	MMX	Shift words in mm left mm/m64 while shifting in 0s.
66 0F F1 /r PSLLW xmm1, xmm2/m128	A	V/V	SSE2	Shift words in xmm1 left by xmm2/m128 while shifting in 0s.
NP 0F 71 /6 ib PSLLW mm1, imm8	B	V/V	MMX	Shift words in mm left by imm8 while shifting in 0s.
66 0F 71 /6 ib PSLLW xmm1, imm8	B	V/V	SSE2	Shift words in xmm1 left by imm8 while shifting in 0s.
NP 0F F2 /r[1] PSLLD mm, mm/m64	A	V/V	MMX	Shift doublewords in mm left by mm/m64 while shifting in 0s.
66 0F F2 /r PSLLD xmm1, xmm2/m128	A	V/V	SSE2	Shift doublewords in xmm1 left by xmm2/m128 while shifting in 0s.
NP 0F 72 /6 ib[1] PSLLD mm, imm8	B	V/V	MMX	Shift doublewords in mm left by imm8 while shifting in 0s.
66 0F 72 /6 ib PSLLD xmm1, imm8	B	V/V	SSE2	Shift doublewords in xmm1 left by imm8 while shifting in 0s.
NP 0F F3 /r[1] PSLLQ mm, mm/m64	A	V/V	MMX	Shift quadword in mm left by mm/m64 while shifting in 0s.
66 0F F3 /r PSLLQ xmm1, xmm2/m128	A	V/V	SSE2	Shift quadwords in xmm1 left by xmm2/m128 while shifting in 0s.
NP 0F 73 /6 ib[1] PSLLQ mm, imm8	B	V/V	MMX	Shift quadword in mm left by imm8 while shifting in 0s.
66 0F 73 /6 ib PSLLQ xmm1, imm8	B	V/V	SSE2	Shift quadwords in xmm1 left by imm8 while shifting in 0s.
VEX.128.66.0F.WIG F1 /r VPSLLW xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 71 /6 ib VPSLLW xmm1, xmm2, imm8	D	V/V	AVX	Shift words in xmm2 left by imm8 while shifting in 0s.
VEX.128.66.0F.WIG F2 /r VPSLLD xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 72 /6 ib VPSLLD xmm1, xmm2, imm8	D	V/V	AVX	Shift doublewords in xmm2 left by imm8 while shifting in 0s.
VEX.128.66.0F.WIG F3 /r VPSLLQ xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 73 /6 ib VPSLLQ xmm1, xmm2, imm8	D	V/V	AVX	Shift quadwords in xmm2 left by imm8 while shifting in 0s.
VEX.256.66.0F.WIG F1 /r VPSLLW ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift words in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 71 /6 ib VPSLLW ymm1, ymm2, imm8	D	V/V	AVX2	Shift words in ymm2 left by imm8 while shifting in 0s.

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
VEX.256.66.0F.WIG F2 /r VPSLLD ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift doublewords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 72 /6 ib VPSLLD ymm1, ymm2, imm8	D	V/V	AVX2	Shift doublewords in ymm2 left by imm8 while shifting in 0s.
VEX.256.66.0F.WIG F3 /r VPSLLQ ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift quadwords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 73 /6 ib VPSLLQ ymm1, ymm2, imm8	D	V/V	AVX2	Shift quadwords in ymm2 left by imm8 while shifting in 0s.
EVEX.128.66.0F.WIG F1 /r VPSLLW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG F1 /r VPSLLW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Shift words in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG F1 /r VPSLLW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW OR AVX10.1[2]	Shift words in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.WIG 71 /6 ib VPSLLW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Shift words in xmm2/m128 left by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG 71 /6 ib VPSLLW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Shift words in ymm2/m256 left by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG 71 /6 ib VPSLLW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW OR AVX10.1[2]	Shift words in zmm2/m512 left by imm8 while shifting in 0 using writemask k1.
EVEX.128.66.0F.W0 F2 /r VPSLLD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.256.66.0F.W0 F2 /r VPSLLD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift doublewords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.512.66.0F.W0 F2 /r VPSLLD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1[2]	Shift doublewords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.128.66.0F.W0 72 /6 ib VPSLLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift doublewords in xmm2/m128/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W0 72 /6 ib VPSLLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift doublewords in ymm2/m256/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W0 72 /6 ib VPSLLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F OR AVX10.1[2]	Shift doublewords in zmm2/m512/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W1 F3 /r VPSLLQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.W1 F3 /r VPSLLQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift quadwords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 F3 /r VPSLLQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1[2]	Shift quadwords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W1 73 /6 ib VPSLLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift quadwords in xmm2/m128/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W1 73 /6 ib VPSLLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift quadwords in ymm2/m256/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 73 /6 ib VPSLLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F OR AVX10.1[2]	Shift quadwords in zmm2/m512/m64bcst left by imm8 while shifting in 0s using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (r, w)	imm8	N/A	N/A
C	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
D	N/A	VEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
F	Full	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
G	Mem128	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 1-17 gives an example of shifting words in a 64-bit operand.

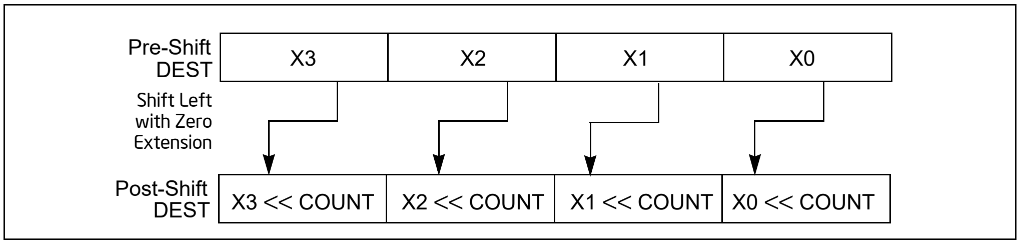


Figure 1-17. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand

The (V)PSLLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the (V)PSLLD instruction shifts each of the doublewords in the destination operand; and the (V)PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. 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 encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /6, or EVEX.128.66.0F 71-73 /6), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSLLW (With 64-bit Operand)

```
    IF (COUNT > 15)
    THEN
        DEST[64:0] := 0000000000000000H;
    ELSE
        DEST[15:0] := ZeroExtend(DEST[15:0] << COUNT);
        (* Repeat shift operation for 2nd and 3rd words *)
        DEST[63:48] := ZeroExtend(DEST[63:48] << COUNT);
    FI;
PSLLD (with 64-bit operand)
    IF (COUNT > 31)
    THEN
        DEST[64:0] := 0000000000000000H;
    ELSE
        DEST[31:0] := ZeroExtend(DEST[31:0] << COUNT);
        DEST[63:32] := ZeroExtend(DEST[63:32] << COUNT);
    FI;
```

PSLLQ (With 64-bit Operand)

```
    IF (COUNT > 63)
    THEN
        DEST[64:0] := 0000000000000000H;
    ELSE
        DEST := ZeroExtend(DEST << COUNT);
    FI;

LOGICAL_LEFT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[15:0] := ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] := ZeroExtend(SRC[127:112] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] := 0
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] << COUNT);
```

```
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] := ZeroExtend(SRC[127:96] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] := 0
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] := ZeroExtend(SRC[127:64] << COUNT);
FI;
LOGICAL_LEFT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
    DEST[255:128] := 00000000000000000000000000000000H
ELSE
    DEST[15:0] := ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] := ZeroExtend(SRC[255:240] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
    DEST[255:128] := 00000000000000000000000000000000H
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[255:224] := ZeroExtend(SRC[255:224] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
    DEST[255:128] := 00000000000000000000000000000000H
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] << COUNT);
```

```
    DEST[127:64] := ZeroExtend(SRC[127:64] << COUNT)
    DEST[191:128] := ZeroExtend(SRC[191:128] << COUNT);
    DEST[255:192] := ZeroExtend(SRC[255:192] << COUNT);
FI;
```

VPSLLW (EVEX Versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

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*            ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSLLW (EVEX Versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

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*            ; zeroing-masking
                    DEST[i+15:i] = 0
```

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

VPSLLW (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSLLW (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_WORD_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSLLW (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSLLW (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSLLW (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLW (xmm, imm8)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSLLD (EVEX versions, imm8)

```
(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] := LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] := LOGICAL_LEFT_SHIFT_DWORDS1(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
```

VPSLLD (EVEX Versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
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
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSLLD (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSLLD (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSLLD (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSLLD (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSLLD (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLD (xmm, imm8)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSLLQ (EVEX Versions, imm8)

```
(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] := LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] := LOGICAL_LEFT_SHIFT_QWORDS1(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
```

VPSLLQ (EVEX Versions, xmm/m128)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
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
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSLLQ (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSLLQ (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSLLQ (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSLLQ (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSLLQ (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLQ (xmm, imm8)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSLLD __m512i _mm512_slli_epi32(__m512i a, unsigned int imm);
VPSLLD __m512i _mm512_mask_slli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m512i _mm512_maskz_slli_epi32( __mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m256i _mm256_mask_slli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m256i _mm256_maskz_slli_epi32( __mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m128i _mm_mask_slli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m128i _mm_maskz_slli_epi32( __mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m512i _mm512_sll_epi32(__m512i a, __m128i cnt);
VPSLLD __m512i _mm512_mask_sll_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m512i _mm512_maskz_sll_epi32( __mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m256i _mm256_mask_sll_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m256i _mm256_maskz_sll_epi32( __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m128i _mm_mask_sll_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLD __m128i _mm_maskz_sll_epi32( __mmask8 k, __m128i a, __m128i cnt);
VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_maskz_slli_epi64( __mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m256i _mm256_mask_slli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m256i _mm256_maskz_slli_epi64( __mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m128i _mm_mask_slli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m128i _mm_maskz_slli_epi64( __mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_maskz_sll_epi64( __mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m256i _mm256_mask_sll_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m256i _mm256_maskz_sll_epi64( __mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m128i _mm_mask_sll_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLQ __m128i _mm_maskz_sll_epi64( __mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m512i _mm512_slli_epi16(__m512i a, unsigned int imm);
VPSLLW __m512i _mm512_mask_slli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m512i _mm512_maskz_slli_epi16( __mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m256i _mm256_mask_slli_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m256i _mm256_maskz_slli_epi16( __mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m128i _mm_mask_slli_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m128i _mm_maskz_slli_epi16( __mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m512i _mm512_sll_epi16(__m512i a, __m128i cnt);
VPSLLW __m512i _mm512_mask_sll_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m512i _mm512_maskz_sll_epi16( __mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m256i _mm256_mask_sll_epi16(__m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSLLW __m256i _mm256_maskz_sll_epi16( __mmask16 k, __m256i a, __m128i cnt);
```

VPSLLW __m128i _mm_mask_sll_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m128i _mm_maskz_sll_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSLLW __m64 _mm_slli_pi16 (__m64 m, int count)
PSLLW __m64 _mm_sll_pi16(__m64 m, __m64 count)
(V)PSLLW __m128i _mm_slli_epi16(__m64 m, int count)
(V)PSLLW __m128i _mm_sll_epi16(__m128i m, __m128i count)
VPSLLW __m256i _mm256_slli_epi16 (__m256i m, int count)
VPSLLW __m256i _mm256_sll_epi16 (__m256i m, __m128i count)
PSLLD __m64 _mm_slli_pi32(__m64 m, int count)
PSLLD __m64 _mm_sll_pi32(__m64 m, __m64 count)
(V)PSLLD __m128i _mm_slli_epi32(__m128i m, int count)
(V)PSLLD __m128i _mm_sll_epi32(__m128i m, __m128i count)
VPSLLD __m256i _mm256_slli_epi32 (__m256i m, int count)
VPSLLD __m256i _mm256_sll_epi32 (__m256i m, __m128i count)
PSLLQ __m64 _mm_slli_si64(__m64 m, int count)
PSLLQ __m64 _mm_sll_si64(__m64 m, __m64 count)
(V)PSLLQ __m128i _mm_slli_epi64(__m128i m, int count)
(V)PSLLQ __m128i _mm_sll_epi64(__m128i m, __m128i count)
VPSLLQ __m256i _mm256_slli_epi64 (__m256i m, int count)
VPSLLQ __m256i _mm256_sll_epi64 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

- VEX-encoded instructions:
  - Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Table 2-21, “Type 4 Class Exception Conditions.”
  - Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Table 2-24, “Type 7 Class Exception Conditions.”
- EVEX-encoded VPSLLW (E in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
- EVEX-encoded VPSLLD/Q:
  - Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
  - Syntax with Full tuple type (F in the operand encoding table), see Table 2-49, “Type E4 Class Exception Conditions.”

PSRAW/PSRAD/PSRAQ—Shift Packed Data Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E1 /r[1] PSRAW mm, mm/m64	A	V/V	MMX	Shift words in mm right by mm/m64 while shifting in sign bits.
66 0F E1 /r PSRAW xmm1, xmm2/m128	A	V/V	SSE2	Shift words in xmm1 right by xmm2/m128 while shifting in sign bits.
NP 0F 71 /4 ib[1] PSRAW mm, imm8	B	V/V	MMX	Shift words in mm right by imm8 while shifting in sign bits
66 0F 71 /4 ib PSRAW xmm1, imm8	B	V/V	SSE2	Shift words in xmm1 right by imm8 while shifting in sign bits
NP 0F E2 /r[1] PSRAD mm, mm/m64	A	V/V	MMX	Shift doublewords in mm right by mm/m64 while shifting in sign bits.
66 0F E2 /r PSRAD xmm1, xmm2/m128	A	V/V	SSE2	Shift doubleword in xmm1 right by xmm2 /m128 while shifting in sign bits.
NP 0F 72 /4 ib[1] PSRAD mm, imm8	B	V/V	MMX	Shift doublewords in mm right by imm8 while shifting in sign bits.
66 0F 72 /4 ib PSRAD xmm1, imm8	B	V/V	SSE2	Shift doublewords in xmm1 right by imm8 while shifting in sign bits.
VEX.128.66.0F.WIG E1 /r VPSRAW xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.128.66.0F.WIG 71 /4 ib VPSRAW xmm1, xmm2, imm8	D	V/V	AVX	Shift words in xmm2 right by imm8 while shifting in sign bits.
VEX.128.66.0F.WIG E2 /r VPSRAD xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.128.66.0F.WIG 72 /4 ib VPSRAD xmm1, xmm2, imm8	D	V/V	AVX	Shift doublewords in xmm2 right by imm8 while shifting in sign bits.
VEX.256.66.0F.WIG E1 /r VPSRAW ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.256.66.0F.WIG 71 /4 ib VPSRAW ymm1, ymm2, imm8	D	V/V	AVX2	Shift words in ymm2 right by imm8 while shifting in sign bits.
VEX.256.66.0F.WIG E2 /r VPSRAD ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.256.66.0F.WIG 72 /4 ib VPSRAD ymm1, ymm2, imm8	D	V/V	AVX2	Shift doublewords in ymm2 right by imm8 while shifting in sign bits.
EVEX.128.66.0F.WIG E1 /r VPSRAW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.WIG E1 /r VPSRAW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F.WIG E1 /r VPSRAW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW OR AVX10.1[2]	Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.WIG 71 /4 ib VPSRAW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Shift words in xmm2/m128 right by imm8 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.WIG 71 /4 ib VPSRAW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Shift words in ymm2/m256 right by imm8 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.WIG 71 /4 ib VPSRAW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW OR AVX10.1[2]	Shift words in zmm2/m512 right by imm8 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W0 E2 /r VPSRAD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W0 E2 /r VPSRAD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W0 E2 /r VPSRAD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1[2]	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W0 72 /4 ib VPSRAD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W0 72 /4 ib VPSRAD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W0 72 /4 ib VPSRAD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F OR AVX10.1[2]	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W1 E2 /r VPSRAQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W1 E2 /r VPSRAQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W1 E2 /r VPSRAQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1[2]	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W1 72 /4 ib VPSRAQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W1 72 /4 ib VPSRAQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W1 72 /4 ib VPSRAQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F OR AVX10.1[2]	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in sign bits using writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (r, w)	imm8	N/A	N/A
C	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
D	N/A	VEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
F	Full	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
G	Mem128	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (words, doublewords or quadwords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for quadwords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 1-18 gives an example of shifting words in a 64-bit operand.)

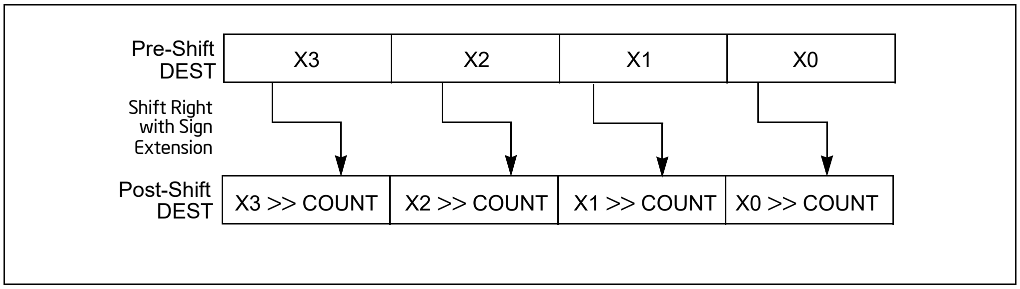


Figure 1-18. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count. If the second source operand is a memory address, 128 bits are loaded.

The (V)PSRAW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand, and the (V)PSRAD instruction shifts each of the doublewords in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. 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 encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /4, EVEX.128.66.0F 71-73 /4), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSRAW (With 64-bit Operand)

```
    IF (COUNT > 15)
        THEN COUNT := 16;
    FI;
    DEST[15:0] := SignExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] := SignExtend(DEST[63:48] >> COUNT);

PSRAD (with 64-bit operand)
    IF (COUNT > 31)
        THEN COUNT := 32;
    FI;
    DEST[31:0] := SignExtend(DEST[31:0] >> COUNT);
    DEST[63:32] := SignExtend(DEST[63:32] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] := SignBit
ELSE
    DEST[31:0] := SignExtend(SRC[31:0] >> COUNT);
FI;

ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] := SignBit
ELSE
```

```
    DEST[63:0] := SignExtend(SRC[63:0] >> COUNT);
FI;

ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN    COUNT := 16;
FI;
DEST[15:0] := SignExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
DEST[255:240] := SignExtend(SRC[255:240] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT := 32;
FI;
DEST[31:0] := SignExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[255:224] := SignExtend(SRC[255:224] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC, VL) ; VL: 128b, 256b or 512b
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
    THEN    COUNT := 64;
FI;
DEST[63:0] := SignExtend(SRC[63:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[VL-1:VL-64] := SignExtend(SRC[VL-1:VL-64] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN    COUNT := 16;
FI;
DEST[15:0] := SignExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[127:112] := SignExtend(SRC[127:112] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT := 32;
FI;
DEST[31:0] := SignExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
DEST[127:96] := SignExtend(SRC[127:96] >> COUNT);
```

VPSRAW (EVEX versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
```

```
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

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*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRAW (EVEX Versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

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*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRAW (ymm, ymm, xmm/m128) - VEX

```
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPSRAW (ymm, imm8) - VEX

```
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
```

```
DEST[MAXVL-1:256] := 0
```

VPSRAW (xmm, xmm, xmm/m128) - VEX

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSRAW (xmm, imm8) - VEX

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSRAW (xmm, xmm, xmm/m128)

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRAW (xmm, imm8)

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRAD (EVEX Versions, imm8)

```
(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] := ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] := ARITHMETIC_RIGHT_SHIFT_DWORDS1(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
```

VPSRAD (EVEX Versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
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
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRAD (ymm, ymm, xmm/m128) - VEX

```
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPSRAD (ymm, imm8) - VEX

```
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0
```

VPSRAD (xmm, xmm, xmm/m128) - VEX

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSRAD (xmm, imm8) - VEX

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSRAD (xmm, xmm, xmm/m128)

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRAD (xmm, imm8)

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRAQ (EVEX Versions, imm8)

```
(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] := ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] := ARITHMETIC_RIGHT_SHIFT_QWORDS1(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
```

VPSRAQ (EVEX Versions, xmm/m128)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[VL-1:0] := ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC1[VL-1:0], SRC2, VL)

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*            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSRAD __m512i _mm512_srai_epi32(__m512i a, unsigned int imm);
VPSRAD __m512i _mm512_mask_srai_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m512i _mm512_maskz_srai_epi32( __mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m256i _mm256_mask_srai_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m256i _mm256_maskz_srai_epi32( __mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m128i _mm_mask_srai_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m128i _mm_maskz_srai_epi32( __mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m512i _mm512_sra_epi32(__m512i a, __m128i cnt);
VPSRAD __m512i _mm512_mask_sra_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m512i _mm512_maskz_sra_epi32( __mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m256i _mm256_mask_sra_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m256i _mm256_maskz_sra_epi32( __mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m128i _mm_mask_sra_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAD __m128i _mm_maskz_sra_epi32( __mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m512i _mm512_srai_epi64(__m512i a, unsigned int imm);
VPSRAQ __m512i _mm512_mask_srai_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm)
VPSRAQ __m512i _mm512_maskz_srai_epi64( __mmask8 k, __m512i a, unsigned int imm)
VPSRAQ __m256i _mm256_mask_srai_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m256i _mm256_maskz_srai_epi64( __mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m128i _mm_mask_srai_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m128i _mm_maskz_srai_epi64( __mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m512i _mm512_sra_epi64(__m512i a, __m128i cnt);
VPSRAQ __m512i _mm512_mask_sra_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt)
VPSRAQ __m512i _mm512_maskz_sra_epi64( __mmask8 k, __m512i a, __m128i cnt)
VPSRAQ __m256i _mm256_mask_sra_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m256i _mm256_maskz_sra_epi64( __mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m128i _mm_mask_sra_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m128i _mm_maskz_sra_epi64( __mmask8 k, __m128i a, __m128i cnt);
VPSRAW __m512i _mm512_srai_epi16(__m512i a, unsigned int imm);
VPSRAW __m512i _mm512_mask_srai_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSRAW __m512i _mm512_maskz_srai_epi16( __mmask32 k, __m512i a, unsigned int imm);
VPSRAW __m256i _mm256_mask_srai_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRAW __m256i _mm256_maskz_srai_epi16( __mmask16 k, __m256i a, unsigned int imm);
VPSRAW __m128i _mm_mask_srai_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
```

```
VPSRAW __m128i _mm_maskz_srai_epi16( __mmask8 k, __m128i a, unsigned int imm);
VPSRAW __m512i _mm512_sra_epi16(__m512i a, __m128i cnt);
VPSRAW __m512i _mm512_mask_sra_epi16(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAW __m512i _mm512_maskz_sra_epi16( __mmask16 k, __m512i a, __m128i cnt);
VPSRAW __m256i _mm256_mask_sra_epi16(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAW __m256i _mm256_maskz_sra_epi16( __mmask8 k, __m256i a, __m128i cnt);
VPSRAW __m128i _mm_mask_sra_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAW __m128i _mm_maskz_sra_epi16( __mmask8 k, __m128i a, __m128i cnt);
PSRAW __m64 _mm_srai_pi16 (__m64 m, int count)
PSRAW __m64 _mm_sra_pi16 (__m64 m, __m64 count)
(V)PSRAW __m128i _mm_srai_epi16(__m128i m, int count)
(V)PSRAW __m128i _mm_sra_epi16(__m128i m, __m128i count)
VPSRAW __m256i _mm256_srai_epi16 (__m256i m, int count)
VPSRAW __m256i _mm256_sra_epi16 (__m256i m, __m128i count)
PSRAD __m64 _mm_srai_pi32 (__m64 m, int count)
PSRAD __m64 _mm_sra_pi32 (__m64 m, __m64 count)
(V)PSRAD __m128i _mm_srai_epi32 (__m128i m, int count)
(V)PSRAD __m128i _mm_sra_epi32 (__m128i m, __m128i count)
VPSRAD __m256i _mm256_srai_epi32 (__m256i m, int count)
VPSRAD __m256i _mm256_sra_epi32 (__m256i m, __m128i count)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

- VEX-encoded instructions:
  - Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Table 2-21, “Type 4 Class Exception Conditions.”
  - Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Table 2-24, “Type 7 Class Exception Conditions.”
- EVEX-encoded VPSRAW (E in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
- EVEX-encoded VPSRAD/Q:
  - Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
  - Syntax with Full tuple type (F in the operand encoding table), see Table 2-49, “Type E4 Class Exception Conditions.”

PSRLDQ—Shift Double Quadword Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /3 ib PSRLDQ xmm1, imm8	A	V/V	SSE2	Shift xmm1 right by imm8 while shifting in 0s.
VEX.128.66.0F.WIG 73 /3 ib VPSRLDQ xmm1, xmm2, imm8	B	V/V	AVX	Shift xmm2 right by imm8 bytes while shifting in 0s.
VEX.256.66.0F.WIG 73 /3 ib VPSRLDQ ymm1, ymm2, imm8	B	V/V	AVX2	Shift ymm1 right by imm8 bytes while shifting in 0s.
EVEX.128.66.0F.WIG 73 /3 ib VPSRLDQ xmm1, xmm2/m128, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Shift xmm2/m128 right by imm8 bytes while shifting in 0s and store result in xmm1.
EVEX.256.66.0F.WIG 73 /3 ib VPSRLDQ ymm1, ymm2/m256, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Shift ymm2/m256 right by imm8 bytes while shifting in 0s and store result in ymm1.
EVEX.512.66.0F.WIG 73 /3 ib VPSRLDQ zmm1, zmm2/m512, imm8	C	V/V	AVX512BW OR AVX10.1[1]	Shift zmm2/m512 right by imm8 bytes while shifting in 0s and store result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (r, w)	imm8	N/A	N/A
B	N/A	VEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
C	Full Mem	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A

Description

Shifts the destination operand (first operand) to the right by the number of bytes specified in the count operand (second operand). The empty high-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a YMM register. The count operand applies to both the low and high 128-bit lanes.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

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. The count operand applies to each 128-bit lanes.

Note: VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

VPSRLDQ (EVEX.512 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] >> (TEMP * 8)
DEST[255:128] := SRC[255:128] >> (TEMP * 8)
DEST[383:256] := SRC[383:256] >> (TEMP * 8)
DEST[511:384] := SRC[511:384] >> (TEMP * 8)
DEST[MAXVL-1:512] := 0;
```

VPSRLDQ (VEX.256 and EVEX.256 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] >> (TEMP * 8)
DEST[255:128] := SRC[255:128] >> (TEMP * 8)
DEST[MAXVL-1:256] := 0;
```

VPSRLDQ (VEX.128 and EVEX.128 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := SRC >> (TEMP * 8)
DEST[MAXVL-1:128] := 0;
```

PSRLDQ (128-bit Legacy SSE Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := DEST >> (TEMP * 8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

```
(V)PSRLDQ __m128i _mm_srli_si128 ( __m128i a, int imm)
VPSRLDQ __m256i _mm256_bsrli_epi128 ( __m256i, const int)
VPSRLDQ __m512i _mm512_bsrli_epi128 ( __m512i, int)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D1 /r[1] PSRLW mm, mm/m64	A	V/V	MMX	Shift words in mm right by amount specified in mm/m64 while shifting in 0s.
66 0F D1 /r PSRLW xmm1, xmm2/m128	A	V/V	SSE2	Shift words in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.
NP 0F 71 /2 ib[1] PSRLW mm, imm8	B	V/V	MMX	Shift words in mm right by imm8 while shifting in 0s.
66 0F 71 /2 ib PSRLW xmm1, imm8	B	V/V	SSE2	Shift words in xmm1 right by imm8 while shifting in 0s.
NP 0F D2 /r[1] PSRLD mm, mm/m64	A	V/V	MMX	Shift doublewords in mm right by amount specified in mm/m64 while shifting in 0s.
66 0F D2 /r PSRLD xmm1, xmm2/m128	A	V/V	SSE2	Shift doublewords in xmm1 right by amount specified in xmm2 /m128 while shifting in 0s.
NP 0F 72 /2 ib[1] PSRLD mm, imm8	B	V/V	MMX	Shift doublewords in mm right by imm8 while shifting in 0s.
66 0F 72 /2 ib PSRLD xmm1, imm8	B	V/V	SSE2	Shift doublewords in xmm1 right by imm8 while shifting in 0s.
NP 0F D3 /r[1] PSRLQ mm, mm/m64	A	V/V	MMX	Shift mm right by amount specified in mm/m64 while shifting in 0s.
66 0F D3 /r PSRLQ xmm1, xmm2/m128	A	V/V	SSE2	Shift quadwords in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.
NP 0F 73 /2 ib[1] PSRLQ mm, imm8	B	V/V	MMX	Shift mm right by imm8 while shifting in 0s.
66 0F 73 /2 ib PSRLQ xmm1, imm8	B	V/V	SSE2	Shift quadwords in xmm1 right by imm8 while shifting in 0s.
VEX.128.66.0F.WIG D1 /r VPSRLW xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 71 /2 ib VPSRLW xmm1, xmm2, imm8	D	V/V	AVX	Shift words in xmm2 right by imm8 while shifting in 0s.
VEX.128.66.0F.WIG D2 /r VPSRLD xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 72 /2 ib VPSRLD xmm1, xmm2, imm8	D	V/V	AVX	Shift doublewords in xmm2 right by imm8 while shifting in 0s.
VEX.128.66.0F.WIG D3 /r VPSRLQ xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 73 /2 ib VPSRLQ xmm1, xmm2, imm8	D	V/V	AVX	Shift quadwords in xmm2 right by imm8 while shifting in 0s.
VEX.256.66.0F.WIG D1 /r VPSRLW ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 71 /2 ib VPSRLW ymm1, ymm2, imm8	D	V/V	AVX2	Shift words in ymm2 right by imm8 while shifting in 0s.

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
VEX.256.66.0F.WIG D2 /r VPSRLD ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 72 /2 ib VPSRLD ymm1, ymm2, imm8	D	V/V	AVX2	Shift doublewords in ymm2 right by imm8 while shifting in 0s.
VEX.256.66.0F.WIG D3 /r VPSRLQ ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 73 /2 ib VPSRLQ ymm1, ymm2, imm8	D	V/V	AVX2	Shift quadwords in ymm2 right by imm8 while shifting in 0s.
EVEX.128.66.0F.WIG D1 /r VPSRLW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG D1 /r VPSRLW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG D1 /r VPSRLW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW OR AVX10.1[2]	Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.WIG 71 /2 ib VPSRLW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Shift words in xmm2/m128 right by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG 71 /2 ib VPSRLW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Shift words in ymm2/m256 right by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG 71 /2 ib VPSRLW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW OR AVX10.1[2]	Shift words in zmm2/m512 right by imm8 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W0 D2 /r VPSRLD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W0 D2 /r VPSRLD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W0 D2 /r VPSRLD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1[2]	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W0 72 /2 ib VPSRLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W0 72 /2 ib VPSRLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W0 72 /2 ib VPSRLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F OR AVX10.1[2]	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W1 D3 /r VPSRLQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.W1 D3 /r VPSRLQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 D3 /r VPSRLQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1[2]	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W1 73 /2 ib VPSRLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W1 73 /2 ib VPSRLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 73 /2 ib VPSRLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F OR AVX10.1[2]	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in 0s using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (r, w)	imm8	N/A	N/A
C	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
D	N/A	VEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
F	Full	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
G	Mem128	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 1-19 gives an example of shifting words in a 64-bit operand.

Note that only the low 64-bits of a 128-bit count operand are checked to compute the count.

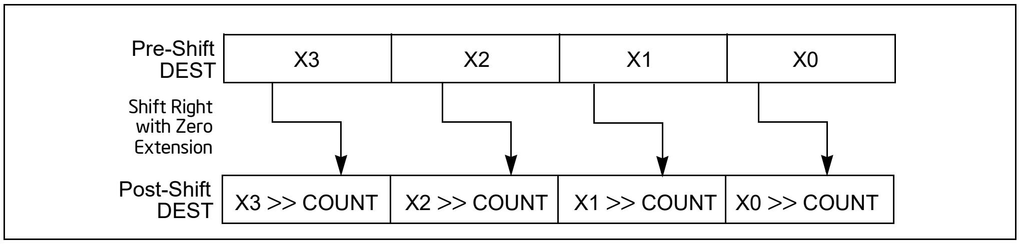


Figure 1-19. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand

The (V)PSRLW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand; the (V)PSRLD instruction shifts each of the doublewords in the destination operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instruction 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. 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 encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /2, or EVEX.128.66.0F 71-73 /2), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSRLW (With 64-bit Operand)

```
IF (COUNT > 15)
THEN
    DEST[64:0] := 0000000000000000H
ELSE
    DEST[15:0] := ZeroExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] := ZeroExtend(DEST[63:48] >> COUNT);
FI;
```

PSRLD (With 64-bit Operand)

```
IF (COUNT > 31)
THEN
```

```
            DEST[64:0] := 0000000000000000H
    ELSE
            DEST[31:0] := ZeroExtend(DEST[31:0] >> COUNT);
            DEST[63:32] := ZeroExtend(DEST[63:32] >> COUNT);
    FI;
```

PSRLQ (With 64-bit Operand)

```
    IF (COUNT > 63)
    THEN
            DEST[64:0] := 0000000000000000H
    ELSE
            DEST := ZeroExtend(DEST >> COUNT);
    FI;
LOGICAL_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] := 0
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] := 0
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[255:0] := 0
ELSE
    DEST[15:0] := ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] := ZeroExtend(SRC[255:240] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[15:0] := ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] := ZeroExtend(SRC[127:112] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
```

```
IF (COUNT > 31)
THEN
    DEST[255:0] := 0
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[255:224] := ZeroExtend(SRC[255:224] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] := ZeroExtend(SRC[127:96] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[255:0] := 0
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] := ZeroExtend(SRC[127:64] >> COUNT);
    DEST[191:128] := ZeroExtend(SRC[191:128] >> COUNT);
    DEST[255:192] := ZeroExtend(SRC[255:192] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] := ZeroExtend(SRC[127:64] >> COUNT);
FI;
```

VPSRLW (EVEX Versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;
```

```
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*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLW (EVEX Versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

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*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLW (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSRLW (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSRLW (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSRLW (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSRLW (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLW (xmm, imm8)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRLD (EVEX Versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
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
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLD (EVEX Versions, imm8)

```
(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] := LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] := LOGICAL_RIGHT_SHIFT_DWORDS1(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
```

VPSRLD (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSRLD (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSRLD (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSRLD (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSRLD (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLD (xmm, imm8)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRLQ (EVEX Versions, xmm/m128)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
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
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLQ (EVEX Versions, imm8)

```
(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] := LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] := LOGICAL_RIGHT_SHIFT_QWORDS1(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
```

VPSRLQ (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSRLQ (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSRLQ (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSRLQ (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSRLQ (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLQ (xmm, imm8)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSRLD __m512i _mm512_srli_epi32(__m512i a, unsigned int imm);
VPSRLD __m512i _mm512_mask_srli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m512i _mm512_maskz_srli_epi32( __mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m256i _mm256_mask_srli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLD __m256i _mm256_maskz_srli_epi32( __mmask8 k, __m256i a, unsigned int imm);
VPSRLD __m128i _mm_mask_srli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m128i _mm_maskz_srli_epi32( __mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m512i _mm512_srl_epi32(__m512i a, __m128i cnt);
VPSRLD __m512i _mm512_mask_srl_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m512i _mm512_maskz_srl_epi32( __mmask16 k, __m512i a, __m128i cnt);
```

VPSRLD __m256i _mm256_mask_srl_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRLD __m256i _mm256_maskz_srl_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSRLD __m128i _mm_mask_srl_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLD __m128i _mm_maskz_srl_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m512i _mm512_srli_epi64(__m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srli_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m256i _mm256_mask_srli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m256i _mm256_maskz_srli_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m128i _mm_mask_srli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m128i _mm_maskz_srli_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m512i _mm512_srl_epi64(__m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m256i _mm256_mask_srl_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m256i _mm256_maskz_srl_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m128i _mm_mask_srl_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m128i _mm_maskz_srl_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m512i _mm512_srli_epi16(__m512i a, unsigned int imm);
VPSRLW __m512i _mm512_mask_srli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m512i _mm512_maskz_srli_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m256i _mm256_mask_srli_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m256i _mm256_maskz_srli_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m128i _mm_mask_srli_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m128i _mm_maskz_srli_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m512i _mm512_srl_epi16(__m512i a, __m128i cnt);
VPSRLW __m512i _mm512_mask_srl_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m512i _mm512_maskz_srl_epi16(__mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m256i _mm256_mask_srl_epi16(__m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSRLW __m256i _mm256_maskz_srl_epi16(__mmask8 k, __mmask16 a, __m128i cnt);
VPSRLW __m128i _mm_mask_srl_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m128i _mm_maskz_srl_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSRLW __m64 _mm_srli_pi16(__m64 m, int count)
PSRLW __m64 _mm_srl_pi16 (__m64 m, __m64 count)
(V)PSRLW __m128i _mm_srli_epi16 (__m128i m, int count)
(V)PSRLW __m128i _mm_srl_epi16 (__m128i m, __m128i count)
VPSRLW __m256i _mm256_srli_epi16 (__m256i m, int count)
VPSRLW __m256i _mm256_srl_epi16 (__m256i m, __m128i count)
PSRLD __m64 _mm_srli_pi32 (__m64 m, int count)
PSRLD __m64 _mm_srl_pi32 (__m64 m, __m64 count)
(V)PSRLD __m128i _mm_srli_epi32 (__m128i m, int count)
(V)PSRLD __m128i _mm_srl_epi32 (__m128i m, __m128i count)
VPSRLD __m256i _mm256_srli_epi32 (__m256i m, int count)
VPSRLD __m256i _mm256_srl_epi32 (__m256i m, __m128i count)
PSRLQ __m64 _mm_srli_si64 (__m64 m, int count)
PSRLQ __m64 _mm_srl_si64 (__m64 m, __m64 count)
(V)PSRLQ __m128i _mm_srli_epi64 (__m128i m, int count)
(V)PSRLQ __m128i _mm_srl_epi64 (__m128i m, __m128i count)
VPSRLQ __m256i _mm256_srli_epi64 (__m256i m, int count)
VPSRLQ __m256i _mm256_srl_epi64 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

- VEX-encoded instructions:
  - Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Table 2-21, “Type 4 Class Exception Conditions.”
  - Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Table 2-24, “Type 7 Class Exception Conditions.”
- EVEX-encoded VPSRLW (E in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
- EVEX-encoded VPSRLD/Q:
  - Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
  - Syntax with Full tuple type (F in the operand encoding table), see Table 2-49, “Type E4 Class Exception Conditions.”

PSUBB/PSUBW/PSUBD—Subtract Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F8 /r[1] PSUBB mm, mm/m64	A	V/V	MMX	Subtract packed byte integers in mm/m64 from packed byte integers in mm.
66 0F F8 /r PSUBB xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed byte integers in xmm2/m128 from packed byte integers in xmm1.
NP 0F F9 /r[1] PSUBW mm, mm/m64	A	V/V	MMX	Subtract packed word integers in mm/m64 from packed word integers in mm.
66 0F F9 /r PSUBW xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed word integers in xmm2/m128 from packed word integers in xmm1.
NP 0F FA /r[1] PSUBD mm, mm/m64	A	V/V	MMX	Subtract packed doubleword integers in mm/m64 from packed doubleword integers in mm.
66 0F FA /r PSUBD xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed doubleword integers in xmm2/mem128 from packed doubleword integers in xmm1.
VEX.128.66.0F.WIG F8 /r VPSUBB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed byte integers in xmm3/m128 from xmm2.
VEX.128.66.0F.WIG F9 /r VPSUBW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed word integers in xmm3/m128 from xmm2.
VEX.128.66.0F.WIG FA /r VPSUBD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed doubleword integers in xmm3/m128 from xmm2.
VEX.256.66.0F.WIG F8 /r VPSUBB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed byte integers in ymm3/m256 from ymm2.
VEX.256.66.0F.WIG F9 /r VPSUBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed word integers in ymm3/m256 from ymm2.
VEX.256.66.0F.WIG FA /r VPSUBD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed doubleword integers in ymm3/m256 from ymm2.
EVEX.128.66.0F.WIG F8 /r VPSUBB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Subtract packed byte integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG F8 /r VPSUBB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Subtract packed byte integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG F8 /r VPSUBB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Subtract packed byte integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.
EVEX.128.66.0F.WIG F9 /r VPSUBW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Subtract packed word integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG F9 /r VPSUBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Subtract packed word integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG F9 /r VPSUBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Subtract packed word integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 FA /r VPSUBD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Subtract packed doubleword integers in xmm3/m128/m32bcst from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.W0 FA /r VPSUBD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Subtract packed doubleword integers in ymm3/m256/m32bcst from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.W0 FA /r VPSUBD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F OR AVX10.1[2]	Subtract packed doubleword integers in zmm3/m512/m32bcst from zmm2 and store in zmm1 using writemask k1

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD subtract of the packed integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The (V)PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The (V)PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 16 bits are written to the destination element.

The (V)PSUBD instruction subtracts packed doubleword integers. When an individual result is too large or too small to be represented in a doubleword, the result is wrapped around and the low 32 bits are written to the destination element.

Note that the (V)PSUBB, (V)PSUBW, and (V)PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBD: 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 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPSUBB/W: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBB (With 64-bit Operands)

```
    DEST[7:0] := DEST[7:0] − SRC[7:0];
    (* Repeat subtract operation for 2nd through 7th byte *)
    DEST[63:56] := DEST[63:56] − SRC[63:56];
```

PSUBW (With 64-bit Operands)

```
    DEST[15:0] := DEST[15:0] − SRC[15:0];
    (* Repeat subtract operation for 2nd and 3rd word *)
    DEST[63:48] := DEST[63:48] − SRC[63:48];
```

PSUBD (With 64-bit Operands)

```
    DEST[31:0] := DEST[31:0] − SRC[31:0];
    DEST[63:32] := DEST[63:32] − SRC[63:32];
```

PSUBD (With 128-bit Operands)

```
    DEST[31:0] := DEST[31:0] − SRC[31:0];
    (* Repeat subtract operation for 2nd and 3rd doubleword *)
    DEST[127:96] := DEST[127:96] − SRC[127:96];
```

VPSUBB (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] := SRC1[i+7:i] - SRC2[i+7:i]
        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] := 0
```

VPSUBW (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] := SRC1[i+15:i] - SRC2[i+15:i]
        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] := 0
```

VPSUBD (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] - SRC2[31:0]
                ELSE DEST[i+31:i] := 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
```

VPSUBB (VEX.256 Encoded Version)

```
DEST[7:0] := SRC1[7:0]-SRC2[7:0]
DEST[15:8] := SRC1[15:8]-SRC2[15:8]
DEST[23:16] := SRC1[23:16]-SRC2[23:16]
DEST[31:24] := SRC1[31:24]-SRC2[31:24]
DEST[39:32] := SRC1[39:32]-SRC2[39:32]
DEST[47:40] := SRC1[47:40]-SRC2[47:40]
DEST[55:48] := SRC1[55:48]-SRC2[55:48]
DEST[63:56] := SRC1[63:56]-SRC2[63:56]
DEST[71:64] := SRC1[71:64]-SRC2[71:64]
DEST[79:72] := SRC1[79:72]-SRC2[79:72]
DEST[87:80] := SRC1[87:80]-SRC2[87:80]
DEST[95:88] := SRC1[95:88]-SRC2[95:88]
DEST[103:96] := SRC1[103:96]-SRC2[103:96]
DEST[111:104] := SRC1[111:104]-SRC2[111:104]
DEST[119:112] := SRC1[119:112]-SRC2[119:112]
DEST[127:120] := SRC1[127:120]-SRC2[127:120]
DEST[135:128] := SRC1[135:128]-SRC2[135:128]
DEST[143:136] := SRC1[143:136]-SRC2[143:136]
```

```
DEST[151:144] := SRC1[151:144]-SRC2[151:144]
DEST[159:152] := SRC1[159:152]-SRC2[159:152]
DEST[167:160] := SRC1[167:160]-SRC2[167:160]
DEST[175:168] := SRC1[175:168]-SRC2[175:168]
DEST[183:176] := SRC1[183:176]-SRC2[183:176]
DEST[191:184] := SRC1[191:184]-SRC2[191:184]
DEST[199:192] := SRC1[199:192]-SRC2[199:192]
DEST[207:200] := SRC1[207:200]-SRC2[207:200]
DEST[215:208] := SRC1[215:208]-SRC2[215:208]
DEST[223:216] := SRC1[223:216]-SRC2[223:216]
DEST[231:224] := SRC1[231:224]-SRC2[231:224]
DEST[239:232] := SRC1[239:232]-SRC2[239:232]
DEST[247:240] := SRC1[247:240]-SRC2[247:240]
DEST[255:248] := SRC1[255:248]-SRC2[255:248]
DEST[MAXVL-1:256] := 0
```

VPSUBB (VEX.128 Encoded Version)

```
DEST[7:0] := SRC1[7:0]-SRC2[7:0]
DEST[15:8] := SRC1[15:8]-SRC2[15:8]
DEST[23:16] := SRC1[23:16]-SRC2[23:16]
DEST[31:24] := SRC1[31:24]-SRC2[31:24]
DEST[39:32] := SRC1[39:32]-SRC2[39:32]
DEST[47:40] := SRC1[47:40]-SRC2[47:40]
DEST[55:48] := SRC1[55:48]-SRC2[55:48]
DEST[63:56] := SRC1[63:56]-SRC2[63:56]
DEST[71:64] := SRC1[71:64]-SRC2[71:64]
DEST[79:72] := SRC1[79:72]-SRC2[79:72]
DEST[87:80] := SRC1[87:80]-SRC2[87:80]
DEST[95:88] := SRC1[95:88]-SRC2[95:88]
DEST[103:96] := SRC1[103:96]-SRC2[103:96]
DEST[111:104] := SRC1[111:104]-SRC2[111:104]
DEST[119:112] := SRC1[119:112]-SRC2[119:112]
DEST[127:120] := SRC1[127:120]-SRC2[127:120]
DEST[MAXVL-1:128] := 0
```

PSUBB (128-bit Legacy SSE Version)

```
DEST[7:0] := DEST[7:0]-SRC[7:0]
DEST[15:8] := DEST[15:8]-SRC[15:8]
DEST[23:16] := DEST[23:16]-SRC[23:16]
DEST[31:24] := DEST[31:24]-SRC[31:24]
DEST[39:32] := DEST[39:32]-SRC[39:32]
DEST[47:40] := DEST[47:40]-SRC[47:40]
DEST[55:48] := DEST[55:48]-SRC[55:48]
DEST[63:56] := DEST[63:56]-SRC[63:56]
DEST[71:64] := DEST[71:64]-SRC[71:64]
DEST[79:72] := DEST[79:72]-SRC[79:72]
DEST[87:80] := DEST[87:80]-SRC[87:80]
DEST[95:88] := DEST[95:88]-SRC[95:88]
DEST[103:96] := DEST[103:96]-SRC[103:96]
DEST[111:104] := DEST[111:104]-SRC[111:104]
DEST[119:112] := DEST[119:112]-SRC[119:112]
DEST[127:120] := DEST[127:120]-SRC[127:120]
DEST[MAXVL-1:128] (Unmodified)
```

VPSUBW (VEX.256 Encoded Version)

```
DEST[15:0] := SRC1[15:0]-SRC2[15:0]
DEST[31:16] := SRC1[31:16]-SRC2[31:16]
DEST[47:32] := SRC1[47:32]-SRC2[47:32]
DEST[63:48] := SRC1[63:48]-SRC2[63:48]
DEST[79:64] := SRC1[79:64]-SRC2[79:64]
DEST[95:80] := SRC1[95:80]-SRC2[95:80]
DEST[111:96] := SRC1[111:96]-SRC2[111:96]
DEST[127:112] := SRC1[127:112]-SRC2[127:112]
DEST[143:128] := SRC1[143:128]-SRC2[143:128]
DEST[159:144] := SRC1[159:144]-SRC2[159:144]
DEST[175:160] := SRC1[175:160]-SRC2[175:160]
DEST[191:176] := SRC1[191:176]-SRC2[191:176]
DEST[207:192] := SRC1207:192]-SRC2[207:192]
DEST[223:208] := SRC1[223:208]-SRC2[223:208]
DEST[239:224] := SRC1[239:224]-SRC2[239:224]
DEST[255:240] := SRC1[255:240]-SRC2[255:240]
DEST[MAXVL-1:256] := 0
```

VPSUBW (VEX.128 Encoded Version)

```
DEST[15:0] := SRC1[15:0]-SRC2[15:0]
DEST[31:16] := SRC1[31:16]-SRC2[31:16]
DEST[47:32] := SRC1[47:32]-SRC2[47:32]
DEST[63:48] := SRC1[63:48]-SRC2[63:48]
DEST[79:64] := SRC1[79:64]-SRC2[79:64]
DEST[95:80] := SRC1[95:80]-SRC2[95:80]
DEST[111:96] := SRC1[111:96]-SRC2[111:96]
DEST[127:112] := SRC1[127:112]-SRC2[127:112]
DEST[MAXVL-1:128] := 0
```

PSUBW (128-bit Legacy SSE Version)

```
DEST[15:0] := DEST[15:0]-SRC[15:0]
DEST[31:16] := DEST[31:16]-SRC[31:16]
DEST[47:32] := DEST[47:32]-SRC[47:32]
DEST[63:48] := DEST[63:48]-SRC[63:48]
DEST[79:64] := DEST[79:64]-SRC[79:64]
DEST[95:80] := DEST[95:80]-SRC[95:80]
DEST[111:96] := DEST[111:96]-SRC[111:96]
DEST[127:112] := DEST[127:112]-SRC[127:112]
DEST[MAXVL-1:128] (Unmodified)
```

VPSUBD (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0]-SRC2[31:0]
DEST[63:32] := SRC1[63:32]-SRC2[63:32]
DEST[95:64] := SRC1[95:64]-SRC2[95:64]
DEST[127:96] := SRC1[127:96]-SRC2[127:96]
DEST[159:128] := SRC1[159:128]-SRC2[159:128]
DEST[191:160] := SRC1[191:160]-SRC2[191:160]
DEST[223:192] := SRC1[223:192]-SRC2[223:192]
DEST[255:224] := SRC1[255:224]-SRC2[255:224]
DEST[MAXVL-1:256] := 0
```

VPSUBD (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0]-SRC2[31:0]
DEST[63:32] := SRC1[63:32]-SRC2[63:32]
DEST[95:64] := SRC1[95:64]-SRC2[95:64]
DEST[127:96] := SRC1[127:96]-SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

PSUBD (128-bit Legacy SSE Version)

```
DEST[31:0] := DEST[31:0]-SRC[31:0]
DEST[63:32] := DEST[63:32]-SRC[63:32]
DEST[95:64] := DEST[95:64]-SRC[95:64]
DEST[127:96] := DEST[127:96]-SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSUBB __m512i _mm512_sub_epi8(__m512i a, __m512i b);
VPSUBB __m512i _mm512_mask_sub_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBB __m512i _mm512_maskz_sub_epi8( __mmask64 k, __m512i a, __m512i b);
VPSUBB __m256i _mm256_mask_sub_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBB __m256i _mm256_maskz_sub_epi8( __mmask32 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_mask_sub_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBB __m128i _mm_maskz_sub_epi8( __mmask16 k, __m128i a, __m128i b);
VPSUBW __m512i _mm512_sub_epi16(__m512i a, __m512i b);
VPSUBW __m512i _mm512_mask_sub_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBW __m512i _mm512_maskz_sub_epi16( __mmask32 k, __m512i a, __m512i b);
VPSUBW __m256i _mm256_mask_sub_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBW __m256i _mm256_maskz_sub_epi16( __mmask16 k, __m256i a, __m256i b);
VPSUBW __m128i _mm_mask_sub_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBW __m128i _mm_maskz_sub_epi16( __mmask8 k, __m128i a, __m128i b);
VPSUBD __m512i _mm512_sub_epi32(__m512i a, __m512i b);
VPSUBD __m512i _mm512_mask_sub_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPSUBD __m512i _mm512_maskz_sub_epi32( __mmask16 k, __m512i a, __m512i b);
VPSUBD __m256i _mm256_mask_sub_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPSUBD __m256i _mm256_maskz_sub_epi32( __mmask8 k, __m256i a, __m256i b);
VPSUBD __m128i _mm_mask_sub_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBD __m128i _mm_maskz_sub_epi32( __mmask8 k, __m128i a, __m128i b);
PSUBB __m64 _mm_sub_pi8(__m64 m1, __m64 m2)
(V)PSUBB __m128i _mm_sub_epi8 ( __m128i a, __m128i b)
VPSUBB __m256i _mm256_sub_epi8 ( __m256i a, __m256i b)
PSUBW __m64 _mm_sub_pi16(__m64 m1, __m64 m2)
(V)PSUBW __m128i _mm_sub_epi16 ( __m128i a, __m128i b)
VPSUBW __m256i _mm256_sub_epi16 ( __m256i a, __m256i b)
PSUBD __m64 _mm_sub_pi32(__m64 m1, __m64 m2)
(V)PSUBD __m128i _mm_sub_epi32 ( __m128i a, __m128i b)
VPSUBD __m256i _mm256_sub_epi32 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

EVEX-encoded VPSUBB/W, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

PSUBQ—Subtract Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F FB /r[1] PSUBQ mm1, mm2/m64	A	V/V	SSE2	Subtract quadword integer in mm1 from mm2 /m64.
66 0F FB /r PSUBQ xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed quadword integers in xmm1 from xmm2 /m128.
VEX.128.66.0F.WIG FB/r VPSUBQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed quadword integers in xmm3/m128 from xmm2.
VEX.256.66.0F.WIG FB /r VPSUBQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed quadword integers in ymm3/m256 from ymm2.
EVEX.128.66.0F.W1 FB /r VPSUBQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Subtract packed quadword integers in xmm3/m128/m64bcst from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.W1 FB /r VPSUBQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Subtract packed quadword integers in ymm3/m256/m64bcst from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.W1 FB/r VPSUBQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1[2]	Subtract packed quadword integers in zmm3/m512/m64bcst from zmm2 and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. When packed quadword operands are used, a SIMD subtract is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the (V)PSUBQ instruction can operate on either unsigned or signed (two’s complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBQ: 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 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBQ (With 64-Bit Operands)

```
    DEST[63:0] := DEST[63:0] − SRC[63:0];
```

PSUBQ (With 128-Bit Operands)

```
    DEST[63:0] := DEST[63:0] − SRC[63:0];
    DEST[127:64] := DEST[127:64] − SRC[127:64];
```

VPSUBQ (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0]-SRC2[63:0]
DEST[127:64] := SRC1[127:64]-SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

VPSUBQ (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0]-SRC2[63:0]
DEST[127:64] := SRC1[127:64]-SRC2[127:64]
DEST[191:128] := SRC1[191:128]-SRC2[191:128]
DEST[255:192] := SRC1[255:192]-SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VPSUBQ (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] - SRC2[63:0]
                ELSE DEST[i+63:i] := 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 Equivalents

VPSUBQ __m512i _mm512_sub_epi64(__m512i a, __m512i b);
VPSUBQ __m512i _mm512_mask_sub_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPSUBQ __m512i _mm512_maskz_sub_epi64(__mmask8 k, __m512i a, __m512i b);
VPSUBQ __m256i _mm256_mask_sub_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPSUBQ __m256i _mm256_maskz_sub_epi64(__mmask8 k, __m256i a, __m256i b);
VPSUBQ __m128i _mm_mask_sub_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBQ __m128i _mm_maskz_sub_epi64(__mmask8 k, __m128i a, __m128i b);
PSUBQ __m64 _mm_sub_si64(__m64 m1, __m64 m2)
(V)PSUBQ __m128i _mm_sub_epi64(__m128i m1, __m128i m2)
VPSUBQ __m256i _mm256_sub_epi64(__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PSUBSB/PSUBSW—Subtract Packed Signed Integers With Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E8 /r[1] PSUBSB mm, mm/m64	A	V/V	MMX	Subtract signed packed bytes in mm/m64 from signed packed bytes in mm and saturate results.
66 0F E8 /r PSUBSB xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed signed byte integers in xmm2/m128 from packed signed byte integers in xmm1 and saturate results.
NP 0F E9 /r[1] PSUBSW mm, mm/m64	A	V/V	MMX	Subtract signed packed words in mm/m64 from signed packed words in mm and saturate results.
66 0F E9 /r PSUBSW xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed signed word integers in xmm2/m128 from packed signed word integers in xmm1 and saturate results.
VEX.128.66.0F.WIG E8 /r VPSUBSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results.
VEX.128.66.0F.WIG E9 /r VPSUBSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results.
VEX.256.66.0F.WIG E8 /r VPSUBSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed signed byte integers in ymm3/m256 from packed signed byte integers in ymm2 and saturate results.
VEX.256.66.0F.WIG E9 /r VPSUBSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results.
EVEX.128.66.0F.WIG E8 /r VPSUBSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG E8 /r VPSUBSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Subtract packed signed byte integers in ymm3/m256 from packed signed byte integers in ymm2 and saturate results and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG E8 /r VPSUBSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Subtract packed signed byte integers in zmm3/m512 from packed signed byte integers in zmm2 and saturate results and store in zmm1 using writemask k1.
EVEX.128.66.0F.WIG E9 /r VPSUBSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG E9 /r VPSUBSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG E9 /r VPSUBSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Subtract packed signed word integers in zmm3/m512 from packed signed word integers in zmm2 and saturate results and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD subtract of the packed signed integers of the source operand (second operand) from the packed signed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

The (V)PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The (V)PSUBSW instruction subtracts packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBSB (With 64-bit Operands)

```
    DEST[7:0] := SaturateToSignedByte (DEST[7:0] – SRC (7:0]);
    (* Repeat subtract operation for 2nd through 7th bytes *)
    DEST[63:56] := SaturateToSignedByte (DEST[63:56] – SRC[63:56] );
```

PSUBSW (With 64-bit Operands)

```
    DEST[15:0] := SaturateToSignedWord (DEST[15:0] – SRC[15:0] );
    (* Repeat subtract operation for 2nd and 7th words *)
    DEST[63:48] := SaturateToSignedWord (DEST[63:48] – SRC[63:48] );
```

VPSUBSB (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] := SaturateToSignedByte (SRC1[i+7:i] - SRC2[i+7:i])
        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] := 0
```

VPSUBSW (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] := SaturateToSignedWord (SRC1[i+15:i] - SRC2[i+15:i])
        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] := 0;
```

VPSUBSB (VEX.256 Encoded Version)

```
DEST[7:0] := SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31th bytes *)
DEST[255:248] := SaturateToSignedByte (SRC1[255:248] - SRC2[255:248]);
DEST[MAXVL-1:256] := 0;
```

VPSUBSB (VEX.128 Encoded Version)

```
DEST[7:0] := SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToSignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[MAXVL-1:128] := 0;
```

PSUBSB (128-bit Legacy SSE Version)

```
DEST[7:0] := SaturateToSignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToSignedByte (DEST[127:120] - SRC[127:120]);
DEST[MAXVL-1:128] (Unmodified);
```

VPSUBSW (VEX.256 Encoded Version)

```
DEST[15:0] := SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[255:240] := SaturateToSignedWord (SRC1[255:240] - SRC2[255:240]);
DEST[MAXVL-1:256] := 0;
```

VPSUBSW (VEX.128 Encoded Version)

```
DEST[15:0] := SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] := SaturateToSignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[MAXVL-1:128] := 0;
```

PSUBSW (128-bit Legacy SSE Version)

```
DEST[15:0] := SaturateToSignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] := SaturateToSignedWord (DEST[127:112] - SRC[127:112]);
DEST[MAXVL-1:128] (Unmodified);
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSUBSB __m512i _mm512_subs_epi8(__m512i a, __m512i b);
VPSUBSB __m512i _mm512_mask_subs_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBSB __m512i _mm512_maskz_subs_epi8( __mmask64 k, __m512i a, __m512i b);
VPSUBSB __m256i _mm256_mask_subs_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBSB __m256i _mm256_maskz_subs_epi8( __mmask32 k, __m256i a, __m256i b);
VPSUBSB __m128i _mm_mask_subs_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBSB __m128i _mm_maskz_subs_epi8( __mmask16 k, __m128i a, __m128i b);
VPSUBSW __m512i _mm512_subs_epi16(__m512i a, __m512i b);
VPSUBSW __m512i _mm512_mask_subs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16( __mmask32 k, __m512i a, __m512i b);
VPSUBSW __m256i _mm256_mask_subs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBSW __m256i _mm256_maskz_subs_epi16( __mmask16 k, __m256i a, __m256i b);
VPSUBSW __m128i _mm_mask_subs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBSW __m128i _mm_maskz_subs_epi16( __mmask8 k, __m128i a, __m128i b);
PSUBSB __m64 _mm_subs_pi8(__m64 m1, __m64 m2)
(V)PSUBSB __m128i _mm_subs_epi8(__m128i m1, __m128i m2)
VPSUBSB __m256i _mm256_subs_epi8(__m256i m1, __m256i m2)
PSUBSW __m64 _mm_subs_pi16(__m64 m1, __m64 m2)
(V)PSUBSW __m128i _mm_subs_epi16(__m128i m1, __m128i m2)
VPSUBSW __m256i _mm256_subs_epi16(__m256i m1, __m256i m2)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers With Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D8 /r[1] PSUBUSB mm, mm/m64	A	V/V	MMX	Subtract unsigned packed bytes in mm/m64 from unsigned packed bytes in mm and saturate result.
66 0F D8 /r PSUBUSB xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed unsigned byte integers in xmm2/m128 from packed unsigned byte integers in xmm1 and saturate result.
NP 0F D9 /r[1] PSUBUSW mm, mm/m64	A	V/V	MMX	Subtract unsigned packed words in mm/m64 from unsigned packed words in mm and saturate result.
66 0F D9 /r PSUBUSW xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed unsigned word integers in xmm2/m128 from packed unsigned word integers in xmm1 and saturate result.
VEX.128.66.0F.WIG D8 /r VPSUBUSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed unsigned byte integers in xmm3/m128 from packed unsigned byte integers in xmm2 and saturate result.
VEX.128.66.0F.WIG D9 /r VPSUBUSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed unsigned word integers in xmm3/m128 from packed unsigned word integers in xmm2 and saturate result.
VEX.256.66.0F.WIG D8 /r VPSUBUSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed unsigned byte integers in ymm3/m256 from packed unsigned byte integers in ymm2 and saturate result.
VEX.256.66.0F.WIG D9 /r VPSUBUSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed unsigned word integers in ymm3/m256 from packed unsigned word integers in ymm2 and saturate result.
EVEX.128.66.0F.WIG D8 /r VPSUBUSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Subtract packed unsigned byte integers in xmm3/m128 from packed unsigned byte integers in xmm2, saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG D8 /r VPSUBUSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Subtract packed unsigned byte integers in ymm3/m256 from packed unsigned byte integers in ymm2, saturate results and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG D8 /r VPSUBUSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Subtract packed unsigned byte integers in zmm3/m512 from packed unsigned byte integers in zmm2, saturate results and store in zmm1 using writemask k1.
EVEX.128.66.0F.WIG D9 /r VPSUBUSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Subtract packed unsigned word integers in xmm3/m128 from packed unsigned word integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG D9 /r VPSUBUSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Subtract packed unsigned word integers in ymm3/m256 from packed unsigned word integers in ymm2, saturate results and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG D9 /r VPSUBUSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Subtract packed unsigned word integers in zmm3/m512 from packed unsigned word integers in zmm2, saturate results and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD subtract of the packed unsigned integers of the source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands.

The (V)PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The (V)PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBUSB (With 64-bit Operands)

```
    DEST[7:0] := SaturateToUnsignedByte (DEST[7:0] – SRC (7:0] );
    (* Repeat add operation for 2nd through 7th bytes *)
    DEST[63:56] := SaturateToUnsignedByte (DEST[63:56] – SRC[63:56];
```

PSUBUSW (With 64-bit Operands)

```
    DEST[15:0] := SaturateToUnsignedWord (DEST[15:0] − SRC[15:0] );
    (* Repeat add operation for 2nd and 3rd words *)
    DEST[63:48] := SaturateToUnsignedWord (DEST[63:48] − SRC[63:48] );
```

VPSUBUSB (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] := SaturateToUnsignedByte (SRC1[i+7:i] - SRC2[i+7:i])
        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] := 0;
```

VPSUBUSW (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] := SaturateToUnsignedWord (SRC1[i+15:i] - SRC2[i+15:i])
        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] := 0;
```

VPSUBUSB (VEX.256 Encoded Version)

```
DEST[7:0] := SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31st bytes *)
DEST[255:148] := SaturateToUnsignedByte (SRC1[255:248] - SRC2[255:248]);
DEST[MAXVL-1:256] := 0;
```

VPSUBUSB (VEX.128 Encoded Version)

```
DEST[7:0] := SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToUnsignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[MAXVL-1:128] := 0
```

PSUBUSB (128-bit Legacy SSE Version)

```
DEST[7:0] := SaturateToUnsignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToUnsignedByte (DEST[127:120] - SRC[127:120]);
DEST[MAXVL-1:128] (Unmodified)
```

VPSUBUSW (VEX.256 Encoded Version)

```
DEST[15:0] := SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[255:240] := SaturateToUnsignedWord (SRC1[255:240] - SRC2[255:240]);
DEST[MAXVL-1:256] := 0;
```

VPSUBUSW (VEX.128 Encoded Version)

```
DEST[15:0] := SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] := SaturateToUnsignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[MAXVL-1:128] := 0
```

PSUBUSW (128-bit Legacy SSE Version)

```
DEST[15:0] := SaturateToUnsignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] := SaturateToUnsignedWord (DEST[127:112] - SRC[127:112]);
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

VPSUBUSB __m512i _mm512_subs_epu8(__m512i a, __m512i b);
VPSUBUSB __m512i _mm512_mask_subs_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m512i _mm512_maskz_subs_epu8(__mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m256i _mm256_mask_subs_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m256i _mm256_maskz_subs_epu8(__mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m128i _mm_mask_subs_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBUSB __m128i _mm_maskz_subs_epu8(__mmask16 k, __m128i a, __m128i b);
VPSUBUSW __m512i _mm512_subs_epu16(__m512i a, __m512i b);
VPSUBUSW __m512i _mm512_mask_subs_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBUSW __m512i _mm512_maskz_subs_epu16(__mmask32 k, __m512i a, __m512i b);
VPSUBUSW __m256i _mm256_mask_subs_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBUSW __m256i _mm256_maskz_subs_epu16(__mmask16 k, __m256i a, __m256i b);
VPSUBUSW __m128i _mm_mask_subs_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBUSW __m128i _mm_maskz_subs_epu16(__mmask8 k, __m128i a, __m128i b);
PSUBUSB __m64 _mm_subs_pu8(__m64 m1, __m64 m2)
(V)PSUBUSB __m128i _mm_subs_epu8(__m128i m1, __m128i m2)
VPSUBUSB __m256i _mm256_subs_epu8(__m256i m1, __m256i m2)
PSUBUSW __m64 _mm_subs_pu16(__m64 m1, __m64 m2)
(V)PSUBUSW __m128i _mm_subs_epu16(__m128i m1, __m128i m2)
VPSUBUSW __m256i _mm256_subs_epu16(__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 68 /r[1] PUNPCKHBW mm, mm/m64	A	V/V	MMX	Unpack and interleave high-order bytes from mm and mm/m64 into mm.
66 0F 68 /r PUNPCKHBW xmm1, xmm2/m128	A	V/V	SSE2	Unpack and interleave high-order bytes from xmm1 and xmm2/m128 into xmm1.
NP 0F 69 /r[1] PUNPCKHWD mm, mm/m64	A	V/V	MMX	Unpack and interleave high-order words from mm and mm/m64 into mm.
66 0F 69 /r PUNPCKHWD xmm1, xmm2/m128	A	V/V	SSE2	Unpack and interleave high-order words from xmm1 and xmm2/m128 into xmm1.
NP 0F 6A /r[1] PUNPCKHDQ mm, mm/m64	A	V/V	MMX	Unpack and interleave high-order doublewords from mm and mm/m64 into mm.
66 0F 6A /r PUNPCKHDQ xmm1, xmm2/m128	A	V/V	SSE2	Unpack and interleave high-order doublewords from xmm1 and xmm2/m128 into xmm1.
66 0F 6D /r PUNPCKHQDQ xmm1, xmm2/m128	A	V/V	SSE2	Unpack and interleave high-order quadwords from xmm1 and xmm2/m128 into xmm1.
VEX.128.66.0F.WIG 68/r VPUNPCKHBW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 69/r VPUNPCKHWD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order words from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 6A/r VPUNPCKHDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order doublewords from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 6D/r VPUNPCKHQDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order quadword from xmm2 and xmm3/m128 into xmm1 register.
VEX.256.66.0F.WIG 68 /r VPUNPCKHBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order bytes from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 69 /r VPUNPCKHWD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order words from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 6A /r VPUNPCKHDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order doublewords from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 6D /r VPUNPCKHQDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order quadword from ymm2 and ymm3/m256 into ymm1 register.
EVEX.128.66.0F.WIG 68 /r VPUNPCKHBW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.
EVEX.128.66.0F.WIG 69 /r VPUNPCKHWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Interleave high-order words from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.
EVEX.128.66.0F.W0 6A /r VPUNPCKHDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Interleave high-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register using k1 write mask.
EVEX.128.66.0F.W1 6D /r VPUNPCKHQDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Interleave high-order quadword from xmm2 and xmm3/m128/m64bcst into xmm1 register using k1 write mask.

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EVEX.256.66.0F.WIG 68 /r VPUNPCKHBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Interleave high-order bytes from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.256.66.0F.WIG 69 /r VPUNPCKHWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Interleave high-order words from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.256.66.0F.W0 6A /r VPUNPCKHDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Interleave high-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register using k1 write mask.
EVEX.256.66.0F.W1 6D /r VPUNPCKHQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Interleave high-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register using k1 write mask.
EVEX.512.66.0F.WIG 68/r VPUNPCKHBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Interleave high-order bytes from zmm2 and zmm3/m512 into zmm1 register.
EVEX.512.66.0F.WIG 69/r VPUNPCKHWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Interleave high-order words from zmm2 and zmm3/m512 into zmm1 register.
EVEX.512.66.0F.W0 6A /r VPUNPCKHDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F OR AVX10.1[2]	Interleave high-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register using k1 write mask.
EVEX.512.66.0F.W1 6D /r VPUNPCKHQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F OR AVX10.1[2]	Interleave high-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register using k1 write mask.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. Figure 1-20 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.

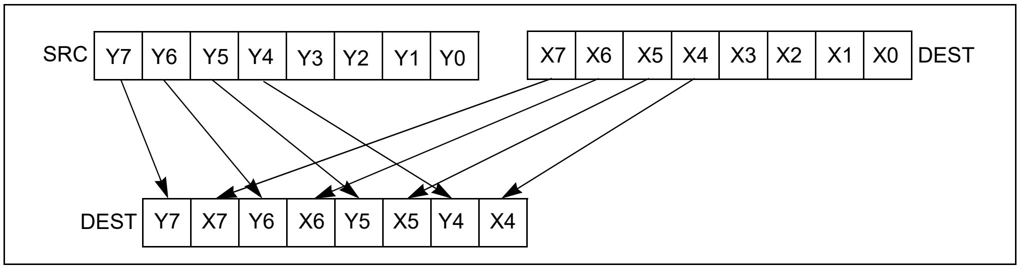


Figure 1-20. PUNPCKHBW Instruction Operation Using 64-bit Operands

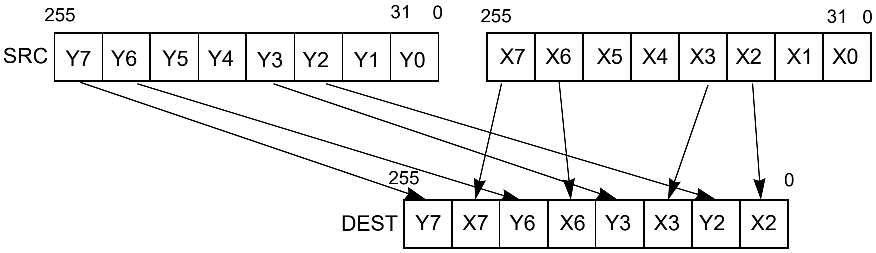


Figure 1-21. 256-bit VPUNPCKHDQ Instruction Operation

When the source data comes from a 64-bit memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the (V)PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the (V)PUNPCKHDQ instruction interleaves the high-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.

EVEX encoded VPUNPCKHDQ/QDQ: 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 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKHWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PUNPCKHBW Instruction With 64-bit Operands:

```
    DEST[7:0] := DEST[39:32];
    DEST[15:8] := SRC[39:32];
    DEST[23:16] := DEST[47:40];
    DEST[31:24] := SRC[47:40];
    DEST[39:32] := DEST[55:48];
    DEST[47:40] := SRC[55:48];
    DEST[55:48] := DEST[63:56];
    DEST[63:56] := SRC[63:56];
```

PUNPCKHW Instruction With 64-bit Operands:

```
    DEST[15:0] := DEST[47:32];
    DEST[31:16] := SRC[47:32];
    DEST[47:32] := DEST[63:48];
    DEST[63:48] := SRC[63:48];
```

PUNPCKHDQ Instruction With 64-bit Operands:

```
    DEST[31:0] := DEST[63:32];
    DEST[63:32] := SRC[63:32];
```

```
INTERLEAVE_HIGH_BYTES_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_HIGH_BYTES_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] := INTERLEAVE_HIGH_BYTES_256b(SRC1[511:256], SRC[511:256])
```

```
INTERLEAVE_HIGH_BYTES_256b (SRC1, SRC2)
DEST[7:0] := SRC1[71:64]
DEST[15:8] := SRC2[71:64]
DEST[23:16] := SRC1[79:72]
DEST[31:24] := SRC2[79:72]
DEST[39:32] := SRC1[87:80]
DEST[47:40] := SRC2[87:80]
DEST[55:48] := SRC1[95:88]
DEST[63:56] := SRC2[95:88]
DEST[71:64] := SRC1[103:96]
DEST[79:72] := SRC2[103:96]
DEST[87:80] := SRC1[111:104]
DEST[95:88] := SRC2[111:104]
DEST[103:96] := SRC1[119:112]
DEST[111:104] := SRC2[119:112]
DEST[119:112] := SRC1[127:120]
DEST[127:120] := SRC2[127:120]
DEST[135:128] := SRC1[199:192]
DEST[143:136] := SRC2[199:192]
DEST[151:144] := SRC1[207:200]
DEST[159:152] := SRC2[207:200]
```

```
DEST[167:160] := SRC1[215:208]
DEST[175:168] := SRC2[215:208]
DEST[183:176] := SRC1[223:216]
DEST[191:184] := SRC2[223:216]
DEST[199:192] := SRC1[231:224]
DEST[207:200] := SRC2[231:224]
DEST[215:208] := SRC1[239:232]
DEST[223:216] := SRC2[239:232]
DEST[231:224] := SRC1[247:240]
DEST[239:232] := SRC2[247:240]
DEST[247:240] := SRC1[255:248]
DEST[255:248] := SRC2[255:248]

INTERLEAVE_HIGH_BYTES (SRC1, SRC2)
DEST[7:0] := SRC1[71:64]
DEST[15:8] := SRC2[71:64]
DEST[23:16] := SRC1[79:72]
DEST[31:24] := SRC2[79:72]
DEST[39:32] := SRC1[87:80]
DEST[47:40] := SRC2[87:80]
DEST[55:48] := SRC1[95:88]
DEST[63:56] := SRC2[95:88]
DEST[71:64] := SRC1[103:96]
DEST[79:72] := SRC2[103:96]
DEST[87:80] := SRC1[111:104]
DEST[95:88] := SRC2[111:104]
DEST[103:96] := SRC1[119:112]
DEST[111:104] := SRC2[119:112]
DEST[119:112] := SRC1[127:120]
DEST[127:120] := SRC2[127:120]

INTERLEAVE_HIGH_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_HIGH_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] := INTERLEAVE_HIGH_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[15:0] := SRC1[79:64]
DEST[31:16] := SRC2[79:64]
DEST[47:32] := SRC1[95:80]
DEST[63:48] := SRC2[95:80]
DEST[79:64] := SRC1[111:96]
DEST[95:80] := SRC2[111:96]
DEST[111:96] := SRC1[127:112]
DEST[127:112] := SRC2[127:112]
DEST[143:128] := SRC1[207:192]
DEST[159:144] := SRC2[207:192]
DEST[175:160] := SRC1[223:208]
DEST[191:176] := SRC2[223:208]
DEST[207:192] := SRC1[239:224]
DEST[223:208] := SRC2[239:224]
DEST[239:224] := SRC1[255:240]
DEST[255:240] := SRC2[255:240]
```

```
INTERLEAVE_HIGH_WORDS (SRC1, SRC2)
DEST[15:0] := SRC1[79:64]
DEST[31:16] := SRC2[79:64]
DEST[47:32] := SRC1[95:80]
DEST[63:48] := SRC2[95:80]
DEST[79:64] := SRC1[111:96]
DEST[95:80] := SRC2[111:96]
DEST[111:96] := SRC1[127:112]
DEST[127:112] := SRC2[127:112]

INTERLEAVE_HIGH_DWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_HIGH_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_HIGH_DWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[127:96] := SRC2[127:96]
DEST[159:128] := SRC1[223:192]
DEST[191:160] := SRC2[223:192]
DEST[223:192] := SRC1[255:224]
DEST[255:224] := SRC2[255:224]

INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[127:96] := SRC2[127:96]

INTERLEAVE_HIGH_QWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_HIGH_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_HIGH_QWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
DEST[191:128] := SRC1[255:192]
DEST[255:192] := SRC2[255:192]

INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
```

PUNPCKHBW (128-bit Legacy SSE Version)

```
DEST[127:0] := INTERLEAVE_HIGH_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKHBW (VEX.128 Encoded Version)

```
DEST[127:0] := INTERLEAVE_HIGH_BYTES(SRC1, SRC2)
DEST[MAXVL-1:127] := 0
```

VPUNPCKHBW (VEX.256 Encoded Version)

```
DEST[255:0] := INTERLEAVE_HIGH_BYTES_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKHBW (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_BYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_BYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TMP_DEST[i+7:i]
        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] := 0
```

PUNPCKHWD (128-bit Legacy SSE Version)

```
DEST[127:0] := INTERLEAVE_HIGH_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKHWD (VEX.128 Encoded Version)

```
DEST[127:0] := INTERLEAVE_HIGH_WORDS(SRC1, SRC2)
DEST[MAXVL-1:127] := 0
```

VPUNPCKHWD (VEX.256 Encoded Version)

```
DEST[255:0] := INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKHWD (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
```

```
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*          ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

PUNPCKHDQ (128-bit Legacy SSE Version)

```
DEST[127:0] := INTERLEAVE_HIGH_DWORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKHDQ (VEX.128 Encoded Version)

```
DEST[127:0] := INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:127] := 0
```

VPUNPCKHDQ (VEX.256 Encoded Version)

```
DEST[255:0] := INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKHDQ (EVEX.512 Encoded Version)

```
(KL, VL) = (4, 128), (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 = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; 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
```

PUNPCKHQDQ (128-bit Legacy SSE Version)

```
DEST[127:0] := INTERLEAVE_HIGH_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

VPUNPCKHQDQ (VEX.128 Encoded Version)

```
DEST[127:0] := INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPUNPCKHQDQ (VEX.256 Encoded Version)

```
DEST[255:0] := INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKHQDQ (EVEX Encoded 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 VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
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*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPUNPCKHBW __m512i _mm512_unpackhi_epi8(__m512i a, __m512i b);
VPUNPCKHBW __m512i _mm512_mask_unpackhi_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);

VPUNPCKHBW __m512i _mm512_maskz_unpackhi_epi8(__mmask64 k, __m512i a, __m512i b);
VPUNPCKHBW __m256i _mm256_mask_unpackhi_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPUNPCKHBW __m256i _mm256_maskz_unpackhi_epi8(__mmask32 k, __m256i a, __m256i b);
VPUNPCKHBW __m128i _mm_mask_unpackhi_epi8(v s, __mmask16 k, __m128i a, __m128i b);
VPUNPCKHBW __m128i _mm_maskz_unpackhi_epi8(__mmask16 k, __m128i a, __m128i b);
VPUNPCKHWD __m512i _mm512_unpackhi_epi16(__m512i a, __m512i b);
VPUNPCKHWD __m512i _mm512_mask_unpackhi_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPUNPCKHWD __m512i _mm512_maskz_unpackhi_epi16(__mmask32 k, __m512i a, __m512i b);
VPUNPCKHWD __m256i _mm256_mask_unpackhi_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKHWD __m256i _mm256_maskz_unpackhi_epi16(__mmask16 k, __m256i a, __m256i b);
VPUNPCKHWD __m128i _mm_mask_unpackhi_epi16(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKHWD __m128i _mm_maskz_unpackhi_epi16(__mmask8 k, __m128i a, __m128i b);
VPUNPCKHDQ __m512i _mm512_unpackhi_epi32(__m512i a, __m512i b);
VPUNPCKHDQ __m512i _mm512_mask_unpackhi_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPUNPCKHDQ __m512i _mm512_maskz_unpackhi_epi32(__mmask16 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i _mm256_mask_unpackhi_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i _mm256_maskz_unpackhi_epi32(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m128i _mm_mask_unpackhi_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m128i _mm_maskz_unpackhi_epi32(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_unpackhi_epi64(__m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m256i _mm256_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m256i _mm256_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m128i _mm_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m128i _mm_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
PUNPCKHBW __m64 _mm_unpackhi_pi8(__m64 m1, __m64 m2)
(V)PUNPCKHBW __m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)
VPUNPCKHBW __m256i _mm256_unpackhi_epi8(__m256i m1, __m256i m2)
PUNPCKHWD __m64 _mm_unpackhi_pi16(__m64 m1,__m64 m2)
(V)PUNPCKHWD __m128i _mm_unpackhi_epi16(__m128i m1,__m128i m2)
VPUNPCKHWD __m256i _mm256_unpackhi_epi16(__m256i m1,__m256i m2)
PUNPCKHDQ __m64 _mm_unpackhi_pi32(__m64 m1, __m64 m2)
(V)PUNPCKHDQ __m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)
VPUNPCKHDQ __m256i _mm256_unpackhi_epi32(__m256i m1, __m256i m2)
(V)PUNPCKHQDQ __m128i _mm_unpackhi_epi64 (__m128i a, __m128i b)
VPUNPCKHQDQ __m256i _mm256_unpackhi_epi64 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPUNPCKHQDQ/QDQ, see Table 2-50, “Type E4NF Class Exception Conditions.”

EVEX-encoded VPUNPCKHBW/WD, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 60 /r[1] PUNPCKLBW mm, mm/m32	A	V/V	MMX	Interleave low-order bytes from mm and mm/m32 into mm.
66 0F 60 /r PUNPCKLBW xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order bytes from xmm1 and xmm2/m128 into xmm1.
NP 0F 61 /r[1] PUNPCKLWD mm, mm/m32	A	V/V	MMX	Interleave low-order words from mm and mm/m32 into mm.
66 0F 61 /r PUNPCKLWD xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order words from xmm1 and xmm2/m128 into xmm1.
NP 0F 62 /r[1] PUNPCKLDQ mm, mm/m32	A	V/V	MMX	Interleave low-order doublewords from mm and mm/m32 into mm.
66 0F 62 /r PUNPCKLDQ xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order doublewords from xmm1 and xmm2/m128 into xmm1.
66 0F 6C /r PUNPCKLQDQ xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order quadword from xmm1 and xmm2/m128 into xmm1 register.
VEX.128.66.0F.WIG 60/r VPUNPCKLBW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 61/r VPUNPCKLWD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order words from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 62/r VPUNPCKLDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order doublewords from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 6C/r VPUNPCKLQDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order quadword from xmm2 and xmm3/m128 into xmm1 register.
VEX.256.66.0F.WIG 60 /r VPUNPCKLBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 61 /r VPUNPCKLWD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 62 /r VPUNPCKLDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order doublewords from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 6C /r VPUNPCKLQDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order quadword from ymm2 and ymm3/m256 into ymm1 register.
EVEX.128.66.0F.WIG 60 /r VPUNPCKLBW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.
EVEX.128.66.0F.WIG 61 /r VPUNPCKLWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Interleave low-order words from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.
EVEX.128.66.0F.W0 62 /r VPUNPCKLDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Interleave low-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register subject to write mask k1.
EVEX.128.66.0F.W1 6C /r VPUNPCKLQDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EVEX.256.66.0F.WIG 60 /r VPUNPCKLBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.256.66.0F.WIG 61 /r VPUNPCKLWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[2]	Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.256.66.0F.W0 62 /r VPUNPCKLDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Interleave low-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register subject to write mask k1.
EVEX.256.66.0F.W1 6C /r VPUNPCKLQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Interleave low-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register subject to write mask k1.
EVEX.512.66.0F.WIG 60/r VPUNPCKLBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Interleave low-order bytes from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.512.66.0F.WIG 61/r VPUNPCKLWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1[2]	Interleave low-order words from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.512.66.0F.W0 62 /r VPUNPCKLDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F OR AVX10.1[2]	Interleave low-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register subject to write mask k1.
EVEX.512.66.0F.W1 6C /r VPUNPCKLQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F OR AVX10.1[2]	Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 23.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 1-22 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

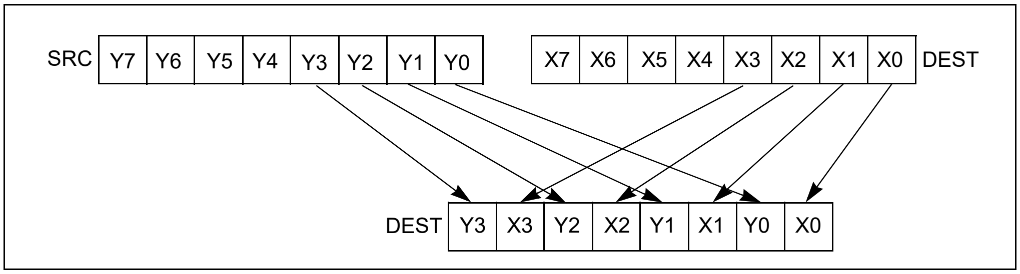


Figure 1-22. PUNPCKLBW Instruction Operation Using 64-bit Operands

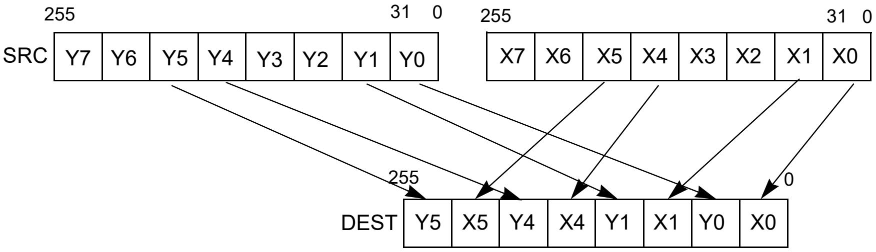


Figure 1-23. 256-bit VPUNPCKLDQ Instruction Operation

When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the (V)PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the (V)PUNPCKLDQ instruction interleaves the low-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 32-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPUNPCKLDQ/QDQ: 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 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKLWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

```
PUNPCKLBW Instruction With 64-bit Operands:
    DEST[63:56] := SRC[31:24];
    DEST[55:48] := DEST[31:24];
    DEST[47:40] := SRC[23:16];
    DEST[39:32] := DEST[23:16];
    DEST[31:24] := SRC[15:8];
    DEST[23:16] := DEST[15:8];
    DEST[15:8] := SRC[7:0];
    DEST[7:0] := DEST[7:0];

PUNPCKLWD Instruction With 64-bit Operands:
    DEST[63:48] := SRC[31:16];
    DEST[47:32] := DEST[31:16];
    DEST[31:16] := SRC[15:0];
    DEST[15:0] := DEST[15:0];

PUNPCKLDQ Instruction With 64-bit Operands:
    DEST[63:32] := SRC[31:0];
    DEST[31:0] := DEST[31:0];
INTERLEAVE_BYTES_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_BYTES_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] := INTERLEAVE_BYTES_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_BYTES_256b (SRC1, SRC2)
DEST[7:0] := SRC1[7:0]
DEST[15:8] := SRC2[7:0]
DEST[23:16] := SRC1[15:8]
DEST[31:24] := SRC2[15:8]
DEST[39:32] := SRC1[23:16]
DEST[47:40] := SRC2[23:16]
DEST[55:48] := SRC1[31:24]
DEST[63:56] := SRC2[31:24]
DEST[71:64] := SRC1[39:32]
DEST[79:72] := SRC2[39:32]
DEST[87:80] := SRC1[47:40]
DEST[95:88] := SRC2[47:40]
DEST[103:96] := SRC1[55:48]
DEST[111:104] := SRC2[55:48]
DEST[119:112] := SRC1[63:56]
DEST[127:120] := SRC2[63:56]
DEST[135:128] := SRC1[135:128]
DEST[143:136] := SRC2[135:128]
DEST[151:144] := SRC1[143:136]
DEST[159:152] := SRC2[143:136]
DEST[167:160] := SRC1[151:144]
DEST[175:168] := SRC2[151:144]
```

```
DEST[183:176] := SRC1[159:152]
DEST[191:184] := SRC2[159:152]
DEST[199:192] := SRC1[167:160]
DEST[207:200] := SRC2[167:160]
DEST[215:208] := SRC1[175:168]
DEST[223:216] := SRC2[175:168]
DEST[231:224] := SRC1[183:176]
DEST[239:232] := SRC2[183:176]
DEST[247:240] := SRC1[191:184]
DEST[255:248] := SRC2[191:184]

INTERLEAVE_BYTES (SRC1, SRC2)
DEST[7:0] := SRC1[7:0]
DEST[15:8] := SRC2[7:0]
DEST[23:16] := SRC1[15:8]
DEST[31:24] := SRC2[15:8]
DEST[39:32] := SRC1[23:16]
DEST[47:40] := SRC2[23:16]
DEST[55:48] := SRC1[31:24]
DEST[63:56] := SRC2[31:24]
DEST[71:64] := SRC1[39:32]
DEST[79:72] := SRC2[39:32]
DEST[87:80] := SRC1[47:40]
DEST[95:88] := SRC2[47:40]
DEST[103:96] := SRC1[55:48]
DEST[111:104] := SRC2[55:48]
DEST[119:112] := SRC1[63:56]
DEST[127:120] := SRC2[63:56]

INTERLEAVE_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] := INTERLEAVE_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[15:0] := SRC1[15:0]
DEST[31:16] := SRC2[15:0]
DEST[47:32] := SRC1[31:16]
DEST[63:48] := SRC2[31:16]
DEST[79:64] := SRC1[47:32]
DEST[95:80] := SRC2[47:32]
DEST[111:96] := SRC1[63:48]
DEST[127:112] := SRC2[63:48]
DEST[143:128] := SRC1[143:128]
DEST[159:144] := SRC2[143:128]
DEST[175:160] := SRC1[159:144]
DEST[191:176] := SRC2[159:144]
DEST[207:192] := SRC1[175:160]
DEST[223:208] := SRC2[175:160]
DEST[239:224] := SRC1[191:176]
DEST[255:240] := SRC2[191:176]

INTERLEAVE_WORDS (SRC1, SRC2)
DEST[15:0] := SRC1[15:0]
DEST[31:16] := SRC2[15:0]
```

```
DEST[47:32] := SRC1[31:16]
DEST[63:48] := SRC2[31:16]
DEST[79:64] := SRC1[47:32]
DEST[95:80] := SRC2[47:32]
DEST[111:96] := SRC1[63:48]
DEST[127:112] := SRC2[63:48]

INTERLEAVE_DWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_DWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[127:96] := SRC2[63:32]
DEST[159:128] := SRC1[159:128]
DEST[191:160] := SRC2[159:128]
DEST[223:192] := SRC1[191:160]
DEST[255:224] := SRC2[191:160]

INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[127:96] := SRC2[63:32]
INTERLEAVE_QWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_QWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[191:128] := SRC1[191:128]
DEST[255:192] := SRC2[191:128]

INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
```

PUNPCKLBW

```
DEST[127:0] := INTERLEAVE_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKLBW (VEX.128 Encoded Instruction)

```
DEST[127:0] := INTERLEAVE_BYTES(SRC1, SRC2)
DEST[MAXVL-1:127] := 0
```

VPUNPCKLBW (VEX.256 Encoded Instruction)

```
DEST[255:0] := INTERLEAVE_BYTES_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKLBW (EVEX.512 Encoded Instruction)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_BYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_BYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TMP_DEST[i+7:i]
        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] := 0
DEST[511:0] := INTERLEAVE_BYTES_512b(SRC1, SRC2)
```

PUNPCKLWD

```
DEST[127:0] := INTERLEAVE_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKLWD (VEX.128 Encoded Instruction)

```
DEST[127:0] := INTERLEAVE_WORDS(SRC1, SRC2)
DEST[MAXVL-1:127] := 0
```

VPUNPCKLWD (VEX.256 Encoded Instruction)

```
DEST[255:0] := INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKLWD (EVEX.512 Encoded Instruction)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

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*          ; zeroing-masking
                        DEST[i+15:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
DEST[511:0] := INTERLEAVE_WORDS_512b(SRC1, SRC2)
```

PUNPCKLDQ

```
DEST[127:0] := INTERLEAVE_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

VPUNPCKLDQ (VEX.128 Encoded Instruction)

```
DEST[127:0] := INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPUNPCKLDQ (VEX.256 Encoded Instruction)

```
DEST[255:0] := INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKLDQ (EVEX Encoded Instructions)

```
(KL, VL) = (4, 128), (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 = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

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

```
ENDFOR
DEST511:0] := INTERLEAVE_DWORDS_512b(SRC1, SRC2)
DEST[MAXVL-1:VL] := 0
```

PUNPCKLQDQ

```
DEST[127:0] := INTERLEAVE_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

VPUNPCKLQDQ (VEX.128 Encoded Instruction)

```
DEST[127:0] := INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPUNPCKLQDQ (VEX.256 Encoded Instruction)

```
DEST[255:0] := INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKLQDQ (EVEX Encoded Instructions)

```
(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 VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
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*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPUNPCKLBW __m512i _mm512_unpacklo_epi8(__m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_mask_unpacklo_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_maskz_unpacklo_epi8( __mmask64 k, __m512i a, __m512i b);
```

```
VPUNPCKLBW __m256i _mm256_mask_unpacklo_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPUNPCKLBW __m256i _mm256_maskz_unpacklo_epi8( __mmask32 k, __m256i a, __m256i b);
VPUNPCKLBW __m128i _mm_mask_unpacklo_epi8(v s, __mmask16 k, __m128i a, __m128i b);
VPUNPCKLBW __m128i _mm_maskz_unpacklo_epi8( __mmask16 k, __m128i a, __m128i b);
VPUNPCKLWD __m512i _mm512_unpacklo_epi16(__m512i a, __m512i b);
VPUNPCKLWD __m512i _mm512_mask_unpacklo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m512i _mm512_maskz_unpacklo_epi16( __mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m256i _mm256_mask_unpacklo_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m256i _mm256_maskz_unpacklo_epi16( __mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m128i _mm_mask_unpacklo_epi16(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLWD __m128i _mm_maskz_unpacklo_epi16( __mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m512i _mm512_unpacklo_epi32(__m512i a, __m512i b);
VPUNPCKLDQ __m512i _mm512_mask_unpacklo_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m512i _mm512_maskz_unpacklo_epi32( __mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m256i _mm256_mask_unpacklo_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m256i _mm256_maskz_unpacklo_epi32( __mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m128i _mm_mask_unpacklo_epi32(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m128i _mm_maskz_unpacklo_epi32( __mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m512i _mm512_unpacklo_epi64(__m512i a, __m512i b);
VPUNPCKLQDQ __m512i _mm512_mask_unpacklo_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m512i _mm512_maskz_unpacklo_epi64( __mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m256i _mm256_mask_unpacklo_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m256i _mm256_maskz_unpacklo_epi64( __mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m128i _mm_mask_unpacklo_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m128i _mm_maskz_unpacklo_epi64( __mmask8 k, __m128i a, __m128i b);
PUNPCKLBW __m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)
(V)PUNPCKLBW __m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)
VPUNPCKLBW __m256i _mm256_unpacklo_epi8 (__m256i m1, __m256i m2)
PUNPCKLWD __m64 _mm_unpacklo_pi16 (__m64 m1, __m64 m2)
(V)PUNPCKLWD __m128i _mm_unpacklo_epi16 (__m128i m1, __m128i m2)
VPUNPCKLWD __m256i _mm256_unpacklo_epi16 (__m256i m1, __m256i m2)
PUNPCKLDQ __m64 _mm_unpacklo_pi32 (__m64 m1, __m64 m2)
(V)PUNPCKLDQ __m128i _mm_unpacklo_epi32 (__m128i m1, __m128i m2)
VPUNPCKLDQ __m256i _mm256_unpacklo_epi32 (__m256i m1, __m256i m2)
(V)PUNPCKLQDQ __m128i _mm_unpacklo_epi64 (__m128i m1, __m128i m2)
VPUNPCKLQDQ __m256i _mm256_unpacklo_epi64 (__m256i m1, __m256i m2)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPUNPCKLDQ/QDQ, see Table 2-50, “Type E4NF Class Exception Conditions.”

EVEX-encoded VPUNPCKLBW/WD, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

PXOR—Logical Exclusive OR

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F EF /r[1] PXOR mm, mm/m64	A	V/V	MMX	Bitwise XOR of mm/m64 and mm.
66 0F EF /r PXOR xmm1, xmm2/m128	A	V/V	SSE2	Bitwise XOR of xmm2/m128 and xmm1.
VEX.128.66.0F.WIG EF /r VPXOR xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise XOR of xmm3/m128 and xmm2.
VEX.256.66.0F.WIG EF /r VPXOR ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Bitwise XOR of ymm3/m256 and ymm2.
EVEX.128.66.0F.W0 EF /r VPXORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise XOR of packed doubleword integers in xmm2 and xmm3/m128 using writemask k1.
EVEX.256.66.0F.W0 EF /r VPXORD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise XOR of packed doubleword integers in ymm2 and ymm3/m256 using writemask k1.
EVEX.512.66.0F.W0 EF /r VPXORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1[2]	Bitwise XOR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.128.66.0F.W1 EF /r VPXORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise XOR of packed quadword integers in xmm2 and xmm3/m128 using writemask k1.
EVEX.256.66.0F.W1 EF /r VPXORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[2]	Bitwise XOR of packed quadword integers in ymm2 and ymm3/m256 using writemask k1.
EVEX.512.66.0F.W1 EF /r VPXORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1[2]	Bitwise XOR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a bitwise logical exclusive-OR (XOR) operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination 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. The upper bits (MAXVL-1:256) of the corresponding register destination are zeroed.

EVEX encoded versions: 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/YMM/XMM register conditionally updated with writemask k1.

Operation

PXOR (64-bit Operand)

```
DEST := DEST XOR SRC
```

PXOR (128-bit Legacy SSE Version)

```
DEST := DEST XOR SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPXOR (VEX.128 Encoded Version)

```
DEST := SRC1 XOR SRC2
DEST[MAXVL-1:128] := 0
```

VPXOR (VEX.256 Encoded Version)

```
DEST := SRC1 XOR SRC2
DEST[MAXVL-1:256] := 0
```

VPXORD (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[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[31:0] := 0
        FI;
   FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPXORQ (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[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

VPXORD __m512i _mm512_xor_epi32(__m512i a, __m512i b)
VPXORD __m512i _mm512_mask_xor_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b)
VPXORD __m512i _mm512_maskz_xor_epi32(__mmask16 m, __m512i a, __m512i b)
VPXORD __m256i _mm256_xor_epi32(__m256i a, __m256i b)
VPXORD __m256i _mm256_mask_xor_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b)
VPXORD __m256i _mm256_maskz_xor_epi32(__mmask8 m, __m256i a, __m256i b)
VPXORD __m128i _mm_xor_epi32(__m128i a, __m128i b)
VPXORD __m128i _mm_mask_xor_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b)
VPXORD __m128i _mm_maskz_xor_epi32(__mmask16 m, __m128i a, __m128i b)
VPXORQ __m512i _mm512_xor_epi64(__m512i a, __m512i b);
VPXORQ __m512i _mm512_mask_xor_epi64(__m512i s, __mmask8 m, __m512i a, __m512i b);
VPXORQ __m512i _mm512_maskz_xor_epi64(__mmask8 m, __m512i a, __m512i b);
VPXORQ __m256i _mm256_xor_epi64(__m256i a, __m256i b);
VPXORQ __m256i _mm256_mask_xor_epi64(__m256i s, __mmask8 m, __m256i a, __m256i b);
VPXORQ __m256i _mm256_maskz_xor_epi64(__mmask8 m, __m256i a, __m256i b);
VPXORQ __m128i _mm_xor_epi64(__m128i a, __m128i b);
VPXORQ __m128i _mm_mask_xor_epi64(__m128i s, __mmask8 m, __m128i a, __m128i b);
VPXORQ __m128i _mm_maskz_xor_epi64(__mmask8 m, __m128i a, __m128i b);
PXOR:__m64 _mm_xor_si64 (__m64 m1, __m64 m2)
(V)PXOR:__m128i _mm_xor_si128 (__m128i a, __m128i b)
VPXOR:__m256i _mm256_xor_si256 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

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

SHUFPD—Packed Interleave Shuffle of Pairs of Double Precision Floating-Point Values

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
66 0F C6 /r ib SHUFPD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle two pairs of double precision floating-point values from xmm1 and xmm2/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.
VEX.128.66.0F.WIG C6 /r ib VSHUFPD xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Shuffle two pairs of double precision floating-point values from xmm2 and xmm3/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.
VEX.256.66.0F.WIG C6 /r ib VSHUFPD ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Shuffle four pairs of double precision floating-point values from ymm2 and ymm3/m256 using imm8 to select from each pair, interleaved result is stored in xmm1.
EVEX.128.66.0F.W1 C6 /r ib VSHUFPD xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Shuffle two pairs of double precision floating-point values from xmm2 and xmm3/m128/m64bcst using imm8 to select from each pair. store interleaved results in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 C6 /r ib VSHUFPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Shuffle four pairs of double precision floating-point values from ymm2 and ymm3/m256/m64bcst using imm8 to select from each pair. store interleaved results in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 C6 /r ib VSHUFPD zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	AVX512F OR AVX10.1[1]	Shuffle eight pairs of double precision floating-point values from zmm2 and zmm3/m512/m64bcst using imm8 to select from each pair. store interleaved results in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Selects a double precision floating-point value of an input pair using a bit control and move to a designated element of the destination operand. The low-to-high order of double precision element of the destination operand is interleaved between the first source operand and the second source operand at the granularity of input pair of 128 bits. Each bit in the imm8 byte, starting from bit 0, is the select control of the corresponding element of the destination to received the shuffled result of an input pair.

EVEX encoded versions: 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 updated according to the writemask. The select controls are the lower 8/4/2 bits of the imm8 byte.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The select controls are the bit 3:0 of the imm8 byte, imm8[7:4) are ignored.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2) are ignored.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination operand and the first source operand is the same and is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2) are ignored.

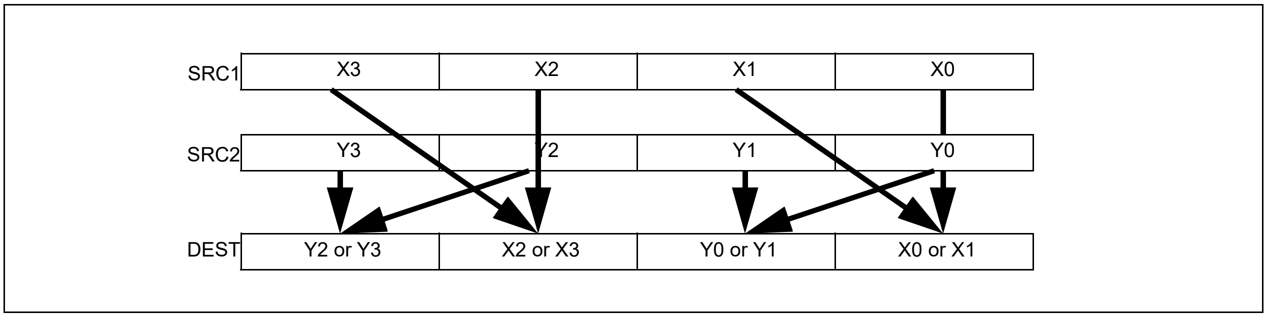


Figure 1-24. 256-bit VSHUFPD Operation of Four Pairs of Double Precision Floating-Point Values

Operation

VSHUFPD (EVEX Encoded Versions When SRC2 is a Vector Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF IMM0[0] = 0
    THEN TMP_DEST[63:0] := SRC1[63:0]
    ELSE TMP_DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN TMP_DEST[127:64] := SRC2[63:0]
    ELSE TMP_DEST[127:64] := SRC2[127:64] FI;
IF VL >= 256
    IF IMM0[2] = 0
        THEN TMP_DEST[191:128] := SRC1[191:128]
        ELSE TMP_DEST[191:128] := SRC1[255:192] FI;
    IF IMM0[3] = 0
        THEN TMP_DEST[255:192] := SRC2[191:128]
        ELSE TMP_DEST[255:192] := SRC2[255:192] FI;
FI;
IF VL >= 512
    IF IMM0[4] = 0
        THEN TMP_DEST[319:256] := SRC1[319:256]
        ELSE TMP_DEST[319:256] := SRC1[383:320] FI;
    IF IMM0[5] = 0
        THEN TMP_DEST[383:320] := SRC2[319:256]
        ELSE TMP_DEST[383:320] := SRC2[383:320] FI;
    IF IMM0[6] = 0
        THEN TMP_DEST[447:384] := SRC1[447:384]
        ELSE TMP_DEST[447:384] := SRC1[511:448] FI;
    IF IMM0[7] = 0
```

```
            THEN TMP_DEST[511:448] := SRC2[447:384]
            ELSE TMP_DEST[511:448] := SRC2[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*              ; zeroing-masking
                        DEST[i+63:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFPD (EVEX Encoded Versions When SRC2 is Memory)

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

```
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1)
            THEN TMP_SRC2[i+63:i] := SRC2[63:0]
            ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF IMM0[0] = 0
    THEN TMP_DEST[63:0] := SRC1[63:0]
    ELSE TMP_DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN TMP_DEST[127:64] := TMP_SRC2[63:0]
    ELSE TMP_DEST[127:64] := TMP_SRC2[127:64] FI;
IF VL >= 256
    IF IMM0[2] = 0
            THEN TMP_DEST[191:128] := SRC1[191:128]
            ELSE TMP_DEST[191:128] := SRC1[255:192] FI;
    IF IMM0[3] = 0
            THEN TMP_DEST[255:192] := TMP_SRC2[191:128]
            ELSE TMP_DEST[255:192] := TMP_SRC2[255:192] FI;
FI;
IF VL >= 512
    IF IMM0[4] = 0
            THEN TMP_DEST[319:256] := SRC1[319:256]
            ELSE TMP_DEST[319:256] := SRC1[383:320] FI;
    IF IMM0[5] = 0
            THEN TMP_DEST[383:320] := TMP_SRC2[319:256]
            ELSE TMP_DEST[383:320] := TMP_SRC2[383:320] FI;
    IF IMM0[6] = 0
            THEN TMP_DEST[447:384] := SRC1[447:384]
            ELSE TMP_DEST[447:384] := SRC1[511:448] FI;
    IF IMM0[7] = 0
            THEN TMP_DEST[511:448] := TMP_SRC2[447:384]
            ELSE TMP_DEST[511:448] := TMP_SRC2[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*            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFPD (VEX.256 Encoded Version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] := SRC1[63:0]
    ELSE DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] := SRC2[63:0]
    ELSE DEST[127:64] := SRC2[127:64] FI;
IF IMM0[2] = 0
    THEN DEST[191:128] := SRC1[191:128]
    ELSE DEST[191:128] := SRC1[255:192] FI;
IF IMM0[3] = 0
    THEN DEST[255:192] := SRC2[191:128]
    ELSE DEST[255:192] := SRC2[255:192] FI;
DEST[MAXVL-1:256] (Unmodified)
```

VSHUFPD (VEX.128 Encoded Version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] := SRC1[63:0]
    ELSE DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] := SRC2[63:0]
    ELSE DEST[127:64] := SRC2[127:64] FI;
DEST[MAXVL-1:128] := 0
```

VSHUFPD (128-bit Legacy SSE Version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] := SRC1[63:0]
    ELSE DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] := SRC2[63:0]
    ELSE DEST[127:64] := SRC2[127:64] FI;
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFPD __m512d _mm512_shuffle_pd(__m512d a, __m512d b, int imm);
VSHUFPD __m512d _mm512_mask_shuffle_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m512d _mm512_maskz_shuffle_pd(__mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m256d _mm256_shuffle_pd (__m256d a, __m256d b, const int select);

VSHUFPD __m256d _mm256_mask_shuffle_pd(__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VSHUFPD __m256d _mm256_maskz_shuffle_pd(__mmask8 k, __m256d a, __m256d b, int imm);
SHUFPD __m128d _mm_shuffle_pd (__m128d a, __m128d b, const int select);
VSHUFPD __m128d _mm_mask_shuffle_pd(__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VSHUFPD __m128d _mm_maskz_shuffle_pd(__mmask8 k, __m128d a, __m128d b, int imm);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”

SHUFPS—Packed Interleave Shuffle of Quadruplets of Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C6 /r ib SHUFPS xmm1, xmm3/m128, imm8	A	V/V	SSE	Select from quadruplet of single precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.128.0F.WIG C6 /r ib VSHUFPS xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Select from quadruplet of single precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.256.0F.WIG C6 /r ib VSHUFPS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Select from quadruplet of single precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1.
EVEX.128.0F.W0 C6 /r ib VSHUFPS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Select from quadruplet of single precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1, subject to writemask k1.
EVEX.256.0F.W0 C6 /r ib VSHUFPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Select from quadruplet of single precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1, subject to writemask k1.
EVEX.512.0F.W0 C6 /r ib VSHUFPS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	C	V/V	AVX512F OR AVX10.1[1]	Select from quadruplet of single precision floating-point values in zmm2 and zmm3/m512 using imm8, interleaved result pairs are stored in zmm1, subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Selects a single precision floating-point value of an input quadruplet using a two-bit control and move to a designated element of the destination operand. Each 64-bit element-pair of a 128-bit lane of the destination operand is interleaved between the corresponding lane of the first source operand and the second source operand at the granularity 128 bits. Each two bits in the imm8 byte, starting from bit 0, is the select control of the corresponding element of a 128-bit lane of the destination to received the shuffled result of an input quadruplet. The two lower elements of a 128-bit lane in the destination receives shuffle results from the quadruple of the first source operand. The next two elements of the destination receives shuffle results from the quadruple of the second source operand.

EVEX encoded versions: 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-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask. imm8[7:0] provides 4 select controls for each applicable 128-bit lane of the destination.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Imm8[7:0] provides 4 select controls for the high and low 128-bit of the destination.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed. Imm8[7:0] provides 4 select controls for each element of the destination.

128-bit Legacy SSE version: The 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 ZMM register destination are unmodified. Imm8[7:0] provides 4 select controls for each element of the destination.

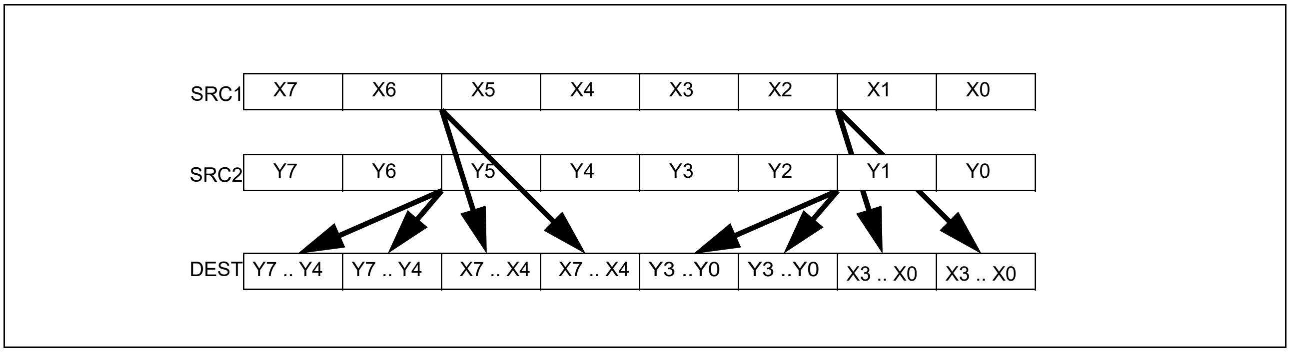


Figure 1-25. 256-bit VSHUFPS Operation of Selection from Input Quadruplet and Pair-wise Interleaved Result

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
}
```

VPSHUFPS (EVEX Encoded Versions When SRC2 is a Vector Register)

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

TMP_DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] := Select4(SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] := Select4(SRC2[127:0], imm8[7:6]);
IF VL >= 256
    TMP_DEST[159:128] := Select4(SRC1[255:128], imm8[1:0]);
    TMP_DEST[191:160] := Select4(SRC1[255:128], imm8[3:2]);
    TMP_DEST[223:192] := Select4(SRC2[255:128], imm8[5:4]);
    TMP_DEST[255:224] := Select4(SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
    TMP_DEST[287:256] := Select4(SRC1[383:256], imm8[1:0]);
    TMP_DEST[319:288] := Select4(SRC1[383:256], imm8[3:2]);
    TMP_DEST[351:320] := Select4(SRC2[383:256], imm8[5:4]);
    TMP_DEST[383:352] := Select4(SRC2[383:256], imm8[7:6]);
    TMP_DEST[415:384] := Select4(SRC1[511:384], imm8[1:0]);
    TMP_DEST[447:416] := Select4(SRC1[511:384], imm8[3:2]);
    TMP_DEST[479:448] := Select4(SRC2[511:384], imm8[5:4]);
    TMP_DEST[511:480] := Select4(SRC2[511:384], imm8[7:6]);
```

```
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; 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
```

VPSHUFPS (EVEX Encoded Versions When SRC2 is Memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1)
        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], imm8[1:0]);
TMP_DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] := Select4(TMP_SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] := Select4(TMP_SRC2[127:0], imm8[7:6]);
IF VL >= 256
    TMP_DEST[159:128] := Select4(SRC1[255:128], imm8[1:0]);
    TMP_DEST[191:160] := Select4(SRC1[255:128], imm8[3:2]);
    TMP_DEST[223:192] := Select4(TMP_SRC2[255:128], imm8[5:4]);
    TMP_DEST[255:224] := Select4(TMP_SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
    TMP_DEST[287:256] := Select4(SRC1[383:256], imm8[1:0]);
    TMP_DEST[319:288] := Select4(SRC1[383:256], imm8[3:2]);
    TMP_DEST[351:320] := Select4(TMP_SRC2[383:256], imm8[5:4]);
    TMP_DEST[383:352] := Select4(TMP_SRC2[383:256], imm8[7:6]);
    TMP_DEST[415:384] := Select4(SRC1[511:384], imm8[1:0]);
    TMP_DEST[447:416] := Select4(SRC1[511:384], imm8[3:2]);
    TMP_DEST[479:448] := Select4(TMP_SRC2[511:384], imm8[5:4]);
    TMP_DEST[511:480] := Select4(TMP_SRC2[511:384], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; 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
```

VSHUFPS (VEX.256 Encoded 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(SRC2[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC2[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(SRC2[255:128], imm8[5:4]);
DEST[255:224] := Select4(SRC2[255:128], imm8[7:6]);
DEST[MAXVL-1:256] := 0
```

VSHUFPS (VEX.128 Encoded 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(SRC2[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC2[127:0], imm8[7:6]);
DEST[MAXVL-1:128] := 0
```

SHUFPS (128-bit Legacy SSE 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(SRC2[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC2[127:0], imm8[7:6]);
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSHUFPS __m512 _mm512_shuffle_ps(__m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_mask_shuffle_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_maskz_shuffle_ps(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int select);
VSHUFPS __m256 _mm256_mask_shuffle_ps(__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VSHUFPS __m256 _mm256_maskz_shuffle_ps(__mmask8 k, __m256 a, __m256 b, int imm);
SHUFPS __m128 _mm_shuffle_ps (__m128 a, __m128 b, const int select);
VSHUFPS __m128 _mm_mask_shuffle_ps(__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VSHUFPS __m128 _mm_maskz_shuffle_ps(__mmask8 k, __m128 a, __m128 b, int imm);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”

SQRTPD—Square Root of Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 51 /r SQRTPD xmm1, xmm2/m128	A	V/V	SSE2	Computes Square Roots of the packed double precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.128.66.0F.WIG 51 /r VSQRTPD xmm1, xmm2/m128	A	V/V	AVX	Computes Square Roots of the packed double precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.256.66.0F.WIG 51 /r VSQRTPD ymm1, ymm2/m256	A	V/V	AVX	Computes Square Roots of the packed double precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.128.66.0F.W1 51 /r VSQRTPD xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Computes Square Roots of the packed double precision floating-point values in xmm2/m128/m64bcst and stores the result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 51 /r VSQRTPD ymm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Computes Square Roots of the packed double precision floating-point values in ymm2/m256/m64bcst and stores the result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 51 /r VSQRTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	Computes Square Roots of the packed double precision floating-point values in zmm2/m512/m64bcst and stores the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a SIMD computation of the square roots of the two, four or eight packed double precision floating-point values in the source operand (the second operand) stores the packed double precision floating-point results in the destination operand (the first operand).

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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

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

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. 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 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 ZMM register destination are unmodified.

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

Operation

VSQRTPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC *is memory*)
                    THEN DEST[i+63:i] := SQRT(SRC[63:0])
                    ELSE DEST[i+63:i] := 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
```

VSQRTPD (VEX.256 Encoded Version)

```
DEST[63:0] := SQRT(SRC[63:0])
DEST[127:64] := SQRT(SRC[127:64])
DEST[191:128] := SQRT(SRC[191:128])
DEST[255:192] := SQRT(SRC[255:192])
DEST[MAXVL-1:256] := 0
.
```

VSQRTPD (VEX.128 Encoded Version)

```
DEST[63:0] := SQRT(SRC[63:0])
DEST[127:64] := SQRT(SRC[127:64])
DEST[MAXVL-1:128] := 0
```

SQRTPD (128-bit Legacy SSE Version)

```
DEST[63:0] := SQRT(SRC[63:0])
DEST[127:64] := SQRT(SRC[127:64])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSQRTPD __m512d _mm512_sqrt_round_pd(__m512d a, int r);
VSQRTPD __m512d _mm512_mask_sqrt_round_pd(__m512d s, __mmask8 k, __m512d a, int r);
VSQRTPD __m512d _mm512_maskz_sqrt_round_pd( __mmask8 k, __m512d a, int r);
VSQRTPD __m256d _mm256_sqrt_pd (__m256d a);
VSQRTPD __m256d _mm256_mask_sqrt_pd(__m256d s, __mmask8 k, __m256d a, int r);
VSQRTPD __m256d _mm256_maskz_sqrt_pd( __mmask8 k, __m256d a, int r);
SQRTPD __m128d _mm_sqrt_pd (__m128d a);
VSQRTPD __m128d _mm_mask_sqrt_pd(__m128d s, __mmask8 k, __m128d a, int r);
VSQRTPD __m128d _mm_maskz_sqrt_pd( __mmask8 k, __m128d a, int r);
```

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions,” additionally:

#UD	If VEX.vvvv != 1111B.

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

#UD	If EVEX.vvvv != 1111B.

SQRTPS—Square Root of Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 51 /r SQRTPS xmm1, xmm2/m128	A	V/V	SSE	Computes Square Roots of the packed single precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.128.0F.WIG 51 /r VSQRTPS xmm1, xmm2/m128	A	V/V	AVX	Computes Square Roots of the packed single precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.256.0F.WIG 51/r VSQRTPS ymm1, ymm2/m256	A	V/V	AVX	Computes Square Roots of the packed single precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.128.0F.W0 51 /r VSQRTPS xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Computes Square Roots of the packed single precision floating-point values in xmm2/m128/m32bcst and stores the result in xmm1 subject to writemask k1.
EVEX.256.0F.W0 51 /r VSQRTPS ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Computes Square Roots of the packed single precision floating-point values in ymm2/m256/m32bcst and stores the result in ymm1 subject to writemask k1.
EVEX.512.0F.W0 51/r VSQRTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	Computes Square Roots of the packed single precision floating-point values in zmm2/m512/m32bcst and stores the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a SIMD computation of the square roots of the four, eight or sixteen packed single precision floating-point values in the source operand (second operand) stores the packed single precision floating-point results in the destination operand.

EVEX.512 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 updated according to the writemask.

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

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. 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 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 ZMM register destination are unmodified.

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

Operation

VSQRTPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC *is memory*)
                    THEN DEST[i+31:i] := SQRT(SRC[31:0])
                    ELSE DEST[i+31:i] := 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
```

VSQRTPS (VEX.256 Encoded Version)

```
DEST[31:0] := SQRT(SRC[31:0])
DEST[63:32] := SQRT(SRC[63:32])
DEST[95:64] := SQRT(SRC[95:64])
DEST[127:96] := SQRT(SRC[127:96])
DEST[159:128] := SQRT(SRC[159:128])
DEST[191:160] := SQRT(SRC[191:160])
DEST[223:192] := SQRT(SRC[223:192])
DEST[255:224] := SQRT(SRC[255:224])
```

VSQRTPS (VEX.128 Encoded Version)

```
DEST[31:0] := SQRT(SRC[31:0])
DEST[63:32] := SQRT(SRC[63:32])
DEST[95:64] := SQRT(SRC[95:64])
DEST[127:96] := SQRT(SRC[127:96])
DEST[MAXVL-1:128] := 0
```

SQRTPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SQRT(SRC[31:0])
DEST[63:32] := SQRT(SRC[63:32])
DEST[95:64] := SQRT(SRC[95:64])
DEST[127:96] := SQRT(SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTPS __m512 _mm512_sqrt_round_ps(__m512 a, int r);
VSQRTPS __m512 _mm512_mask_sqrt_round_ps(__m512 s, __mmask16 k, __m512 a, int r);
VSQRTPS __m512 _mm512_maskz_sqrt_round_ps(__mmask16 k, __m512 a, int r);
VSQRTPS __m256 _mm256_sqrt_ps (__m256 a);
VSQRTPS __m256 _mm256_mask_sqrt_ps(__m256 s, __mmask8 k, __m256 a, int r);
VSQRTPS __m256 _mm256_maskz_sqrt_ps(__mmask8 k, __m256 a, int r);
SQRTPS __m128 _mm_sqrt_ps (__m128 a);
VSQRTPS __m128 _mm_mask_sqrt_ps(__m128 s, __mmask8 k, __m128 a, int r);
VSQRTPS __m128 _mm_maskz_sqrt_ps(__mmask8 k, __m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions,” additionally:

#UD	If VEX.vvvv != 1111B.

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

#UD	If EVEX.vvvv != 1111B.

SQRTSD—Compute Square Root of Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 51/r SQRTSD xmm1,xmm2/m64	A	V/V	SSE2	Computes square root of the low double precision floating-point value in xmm2/m64 and stores the results in xmm1.
VEX.LIG.F2.0F.WIG 51/r VSQRTSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Computes square root of the low double precision floating-point value in xmm3/m64 and stores the results in xmm1. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].
EVEX.LLIG.F2.0F.W1 51/r VSQRTSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F OR AVX10.1[1]	Computes square root of the low double precision floating-point value in xmm3/m64 and stores the results in xmm1 under writemask k1. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Computes the square root of the low double precision floating-point value in the second source operand and stores the double precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. The quadword at bits 127:64 of the destination operand remains unchanged. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:64 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

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

Software should ensure VSQRTSD is encoded with VEX.L=0. Encoding VSQRTSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSQRTSD (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := 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
```

VSQRTSD (VEX.128 Encoded Version)

```
DEST[63:0] := SQRT(SRC2[63:0])
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

SQRTSD (128-bit Legacy SSE Version)

```
DEST[63:0] := SQRT(SRC[63:0])
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSQRTSD __m128d _mm_sqrt_round_sd(__m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_mask_sqrt_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_maskz_sqrt_round_sd(__mmask8 k, __m128d a, __m128d b, int r);
SQRTSD __m128d _mm_sqrt_sd (__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

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

SQRTSS—Compute Square Root of Scalar Single Precision Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 51 /r SQRTSS xmm1, xmm2/m32	A	V/V	SSE	Computes square root of the low single precision floating-point value in xmm2/m32 and stores the results in xmm1.
VEX.LIG.F3.0F.WIG 51 /r VSQRTSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Computes square root of the low single precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].
EVEX.LLIG.F3.0F.W0 51 /r VSQRTSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F OR AVX10.1[1]	Computes square root of the low single precision floating-point value in xmm3/m32 and stores the results in xmm1 under writemask k1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Computes the square root of the low single precision floating-point value in the second source operand and stores the single precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands is an XMM register.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed.

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

Software should ensure VSQRTSS is encoded with VEX.L=0. Encoding VSQRTSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSQRTSS (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SQRT(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
```

VSQRTSS (VEX.128 Encoded Version)

```
DEST[31:0] := SQRT(SRC2[31:0])
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

SQRTSS (128-bit Legacy SSE Version)

```
DEST[31:0] := SQRT(SRC2[31:0])
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSQRTSS __m128 _mm_sqrt_round_ss(__m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_mask_sqrt_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_maskz_sqrt_round_ss( __mmask8 k, __m128 a, __m128 b, int r);
SQRTSS __m128 _mm_sqrt_ss(__m128 a)
```

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

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

SUBPD—Subtract Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5C /r SUBPD xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed double precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.128.66.0F.WIG 5C /r VSUBPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Subtract packed double precision floating-point values in xmm3/mem from xmm2 and store result in xmm1.
VEX.256.66.0F.WIG 5C /r VSUBPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Subtract packed double precision floating-point values in ymm3/mem from ymm2 and store result in ymm1.
EVEX.128.66.0F.W1 5C /r VSUBPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Subtract packed double precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.256.66.0F.W1 5C /r VSUBPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Subtract packed double precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.512.66.0F.W1 5C /r VSUBPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F OR AVX10.1[1]	Subtract packed double precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a SIMD subtract of the two, four or eight packed double precision floating-point values of the second Source operand from the first Source operand, and stores the packed double precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

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

VSUBPD (EVEX Encoded Versions When SRC2 Operand is a Vector Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC1[i+63:i] - SRC2[i+63:i]
    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
```

VSUBPD (EVEX Encoded Versions When SRC2 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] := SRC1[i+63:i] - SRC2[63:0];
                ELSE    EST[i+63:i] := SRC1[i+63:i] - SRC2[i+63:i];
            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
```

VSUBPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] - SRC2[63:0]
DEST[127:64] := SRC1[127:64] - SRC2[127:64]
DEST[191:128] := SRC1[191:128] - SRC2[191:128]
DEST[255:192] := SRC1[255:192] - SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VSUBPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] - SRC2[63:0]
DEST[127:64] := SRC1[127:64] - SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

SUBPD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] - SRC[63:0]
DEST[127:64] := DEST[127:64] - SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSUBPD __m512d _mm512_sub_pd (__m512d a, __m512d b);
VSUBPD __m512d _mm512_mask_sub_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VSUBPD __m512d _mm512_maskz_sub_pd (__mmask8 k, __m512d a, __m512d b);
VSUBPD __m512d _mm512_sub_round_pd (__m512d a, __m512d b, int);
VSUBPD __m512d _mm512_mask_sub_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VSUBPD __m512d _mm512_maskz_sub_round_pd (__mmask8 k, __m512d a, __m512d b, int);
VSUBPD __m256d _mm256_sub_pd (__m256d a, __m256d b);
VSUBPD __m256d _mm256_mask_sub_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VSUBPD __m256d _mm256_maskz_sub_pd (__mmask8 k, __m256d a, __m256d b);
SUBPD __m128d _mm_sub_pd (__m128d a, __m128d b);
VSUBPD __m128d _mm_mask_sub_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBPD __m128d _mm_maskz_sub_pd (__mmask8 k, __m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

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

SUBPS—Subtract Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5C /r SUBPS xmm1, xmm2/m128	A	V/V	SSE	Subtract packed single precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.128.0F.WIG 5C /r VSUBPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Subtract packed single precision floating-point values in xmm3/mem from xmm2 and stores result in xmm1.
VEX.256.0F.WIG 5C /r VSUBPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Subtract packed single precision floating-point values in ymm3/mem from ymm2 and stores result in ymm1.
EVEX.128.0F.W0 5C /r VSUBPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Subtract packed single precision floating-point values from xmm3/m128/m32bcst to xmm2 and stores result in xmm1 with writemask k1.
EVEX.256.0F.W0 5C /r VSUBPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Subtract packed single precision floating-point values from ymm3/m256/m32bcst to ymm2 and stores result in ymm1 with writemask k1.
EVEX.512.0F.W0 5C /r VSUBPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	C	V/V	AVX512F OR AVX10.1[1]	Subtract packed single precision floating-point values in zmm3/m512/m32bcst from zmm2 and stores result in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs a SIMD subtract of the packed single precision floating-point values in the second Source operand from the First Source operand, and stores the packed single precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: 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 first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

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

VSUBPS (EVEX Encoded Versions When SRC2 Operand is a Vector Register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC1[i+31:i] - SRC2[i+31:i]
    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
```

VSUBPS (EVEX Encoded Versions When SRC2 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] := SRC1[i+31:i] - SRC2[31:0];
                ELSE    DEST[i+31:i] := SRC1[i+31:i] - SRC2[i+31:i];
            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
```

VSUBPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[63:32] := SRC1[63:32] - SRC2[63:32]
DEST[95:64] := SRC1[95:64] - SRC2[95:64]
DEST[127:96] := SRC1[127:96] - SRC2[127:96]
DEST[159:128] := SRC1[159:128] - SRC2[159:128]
DEST[191:160] := SRC1[191:160] - SRC2[191:160]
DEST[223:192] := SRC1[223:192] - SRC2[223:192]
DEST[255:224] := SRC1[255:224] - SRC2[255:224].
DEST[MAXVL-1:256] := 0
```

VSUBPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[63:32] := SRC1[63:32] - SRC2[63:32]
DEST[95:64] := SRC1[95:64] - SRC2[95:64]
DEST[127:96] := SRC1[127:96] - SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

SUBPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[63:32] := SRC1[63:32] - SRC2[63:32]
DEST[95:64] := SRC1[95:64] - SRC2[95:64]
DEST[127:96] := SRC1[127:96] - SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSUBPS __m512 _mm512_sub_ps (__m512 a, __m512 b);
VSUBPS __m512 _mm512_mask_sub_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VSUBPS __m512 _mm512_maskz_sub_ps (__mmask16 k, __m512 a, __m512 b);
VSUBPS __m512 _mm512_sub_round_ps (__m512 a, __m512 b, int);
VSUBPS __m512 _mm512_mask_sub_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VSUBPS __m512 _mm512_maskz_sub_round_ps (__mmask16 k, __m512 a, __m512 b, int);
VSUBPS __m256 _mm256_sub_ps (__m256 a, __m256 b);
VSUBPS __m256 _mm256_mask_sub_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VSUBPS __m256 _mm256_maskz_sub_ps (__mmask16 k, __m256 a, __m256 b);
SUBPS __m128 _mm_sub_ps (__m128 a, __m128 b);
VSUBPS __m128 _mm_mask_sub_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBPS __m128 _mm_maskz_sub_ps (__mmask16 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

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

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

SUBSD—Subtract Scalar Double Precision Floating-Point Value

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
F2 0F 5C /r SUBSD xmm1, xmm2/m64	A	V/V	SSE2	Subtract the low double precision floating-point value in xmm2/m64 from xmm1 and store the result in xmm1.
VEX.LIG.F2.0F.WIG 5C /r VSUBSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Subtract the low double precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1.
EVEX.LLIG.F2.0F.W1 5C /r VSUBSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F OR AVX10.1[1]	Subtract the low double precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1 under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Subtract the low double precision floating-point value in the second source operand from the first source operand and stores the double precision floating-point result in the low quadword of the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: 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 encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSD is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSUBSD (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC1[63:0] - SRC2[63:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
```

```
            ELSE                            ; zeoring-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VSUBSD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] - SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

SUBSD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] - SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSUBSD __m128d _mm_mask_sub_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_maskz_sub_sd (__mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_sub_round_sd (__m128d a, __m128d b, int);
VSUBSD __m128d _mm_mask_sub_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSUBSD __m128d _mm_maskz_sub_round_sd (__mmask8 k, __m128d a, __m128d b, int);
SUBSD __m128d _mm_sub_sd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

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

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

SUBSS—Subtract Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5C /r SUBSS xmm1, xmm2/m32	A	V/V	SSE	Subtract the low single precision floating-point value in xmm2/m32 from xmm1 and store the result in xmm1.
VEX.LIG.F3.0F.WIG 5C /r VSUBSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Subtract the low single precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1.
EVEX.LLIG.F3.0F.W0 5C /r VSUBSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F OR AVX10.1[1]	Subtract the low single precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1 under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Subtract the low single precision floating-point value from the second source operand and the first source operand and store the double precision floating-point result in the low doubleword of the destination operand.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: 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 encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSS is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSUBSS (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC1[31:0] - SRC2[31:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
```

```
            ELSE                                ; zoeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VSUBSS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

SUBSS (128-bit Legacy SSE Version)

```
DEST[31:0] := DEST[31:0] - SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSUBSS __m128 _mm_mask_sub_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_maskz_sub_ss (__mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_sub_round_ss (__m128 a, __m128 b, int);
VSUBSS __m128 _mm_mask_sub_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSUBSS __m128 _mm_maskz_sub_round_ss (__mmask8 k, __m128 a, __m128 b, int);
SUBSS __m128 _mm_sub_ss (__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

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

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

UCOMISD—Unordered Compare Scalar Double Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
66 0F 2E /r UCOMISD xmm1, xmm2/m64	A	V/V	SSE2	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2E /r VUCOMISD xmm1, xmm2/m64	A	V/V	AVX	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LLIG.66.0F.W1 2E /r VUCOMISD xmm1, xmm2/m64{sae}	B	V/V	AVX512F OR AVX10.1[1]	Compare low double precision floating-point values in xmm1 and xmm2/m64 and set the EFLAGS flags accordingly.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs an unordered compare of the double precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory

location.

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

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

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

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)UCOMISD (All Versions)

```
RESULT := UnorderedCompare(DEST[63:0] <> SRC[63:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF := 111;
    GREATER_THAN: ZF,PF,CF := 000;
    LESS_THAN: ZF,PF,CF := 001;
    EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VUCOMISD int _mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
UCOMISD int _mm_ucomieq_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomilt_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomile_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomigt_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomige_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomineq_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions,” additionally:

#UD	If VEX.vvvv != 1111B.

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

UCOMISS—Unordered Compare Scalar Single Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2E /r UCOMISS xmm1, xmm2/m32	A	V/V	SSE	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.0F.WIG 2E /r VUCOMISS xmm1, xmm2/m32	A	V/V	AVX	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LLIG.0F.W0 2E /r VUCOMISS xmm1, xmm2/m32{sae}	B	V/V	AVX512F OR AVX10.1[1]	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Compares the single precision floating-point values in the low doublewords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

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

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

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

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)UCOMISS (All Versions)

```
RESULT := UnorderedCompare(DEST[31:0] <> SRC[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF := 111;
    GREATER_THAN: ZF,PF,CF := 000;
    LESS_THAN: ZF,PF,CF := 001;
    EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VUCOMISS	int _mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
UCOMISS	int _mm_ucomieq_ss(__m128 a, __m128 b);
UCOMISS	int _mm_ucomilt_ss(__m128 a, __m128 b);
UCOMISS	int _mm_ucomile_ss(__m128 a, __m128 b);
UCOMISS	int _mm_ucomigt_ss(__m128 a, __m128 b);
UCOMISS	int _mm_ucomige_ss(__m128 a, __m128 b);
UCOMISS	int _mm_ucomineq_ss(__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Invalid (if SNaN Operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions,” additionally:

#UD If VEX.vvvv != 1111B.

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

UNPCKHPD—Unpack and Interleave High Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 15 /r UNPCKHPD xmm1, xmm2/m128	A	V/V	SSE2	Unpacks and Interleaves double precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.128.66.0F.WIG 15 /r VUNPCKHPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.256.66.0F.WIG 15 /r VUNPCKHPD ymm1,ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.128.66.0F.W1 15 /r VUNPCKHPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Unpacks and Interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 15 /r VUNPCKHPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Unpacks and Interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 15 /r VUNPCKHPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1[1]	Unpacks and Interleaves double precision floating-point values from high quadwords of zmm2 and zmm3/m512/m64bcst subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs an interleaved unpack of the high double precision floating-point values from the first source operand and the second source operand. See Figure 4-15 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B.

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 ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

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

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

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 64-bit memory location. The destination operand is a ZMM register, conditionally updated using 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 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second 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.

Operation

VUNPCKHPD (EVEX Encoded Versions When SRC2 is a Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL >= 128
    TMP_DEST[63:0] := SRC1[127:64]
    TMP_DEST[127:64] := SRC2[127:64]
FI;
IF VL >= 256
    TMP_DEST[191:128] := SRC1[255:192]
    TMP_DEST[255:192] := SRC2[255:192]
FI;
IF VL >= 512
    TMP_DEST[319:256] := SRC1[383:320]
    TMP_DEST[383:320] := SRC2[383:320]
    TMP_DEST[447:384] := SRC1[511:448]
    TMP_DEST[511:448] := SRC2[511:448]
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
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKHPD (EVEX Encoded Version When SRC2 is Memory)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[63:0] := SRC1[127:64]
    TMP_DEST[127:64] := TMP_SRC2[127:64]
FI;
IF VL >= 256
    TMP_DEST[191:128] := SRC1[255:192]
    TMP_DEST[255:192] := TMP_SRC2[255:192]
FI;
IF VL >= 512
    TMP_DEST[319:256] := SRC1[383:320]
    TMP_DEST[383:320] := TMP_SRC2[383:320]
    TMP_DEST[447:384] := SRC1[511:448]
    TMP_DEST[511:448] := TMP_SRC2[511:448]
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
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKHPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
DEST[191:128] := SRC1[255:192]
DEST[255:192] := SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VUNPCKHPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

UNPCKHPD (128-bit Legacy SSE Version)

```
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKHPD __m512d _mm512_unpackhi_pd(__m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_mask_unpackhi_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_maskz_unpackhi_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m256d _mm256_unpackhi_pd(__m256d a, __m256d b)
VUNPCKHPD __m256d _mm256_mask_unpackhi_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKHPD __m256d _mm256_maskz_unpackhi_pd(__mmask8 k, __m256d a, __m256d b);
UNPCKHPD __m128d _mm_unpackhi_pd(__m128d a, __m128d b)
VUNPCKHPD __m128d _mm_mask_unpackhi_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKHPD __m128d _mm_maskz_unpackhi_pd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-50, “Type E4NF Class Exception Conditions.”

UNPCKHPS—Unpack and Interleave High Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 15 /r UNPCKHPS xmm1, xmm2/m128	A	V/V	SSE	Unpacks and Interleaves single precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.128.0F.WIG 15 /r VUNPCKHPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves single precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.256.0F.WIG 15 /r VUNPCKHPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves single precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.128.0F.W0 15 /r VUNPCKHPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Unpacks and Interleaves single precision floating-point values from high quadwords of xmm2 and xmm3/m128/m32bcst and write result to xmm1 subject to writemask k1.
EVEX.256.0F.W0 15 /r VUNPCKHPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Unpacks and Interleaves single precision floating-point values from high quadwords of ymm2 and ymm3/m256/m32bcst and write result to ymm1 subject to writemask k1.
EVEX.512.0F.W0 15 /r VUNPCKHPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1[1]	Unpacks and Interleaves single precision floating-point values from high quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs an interleaved unpack of the high single precision floating-point values from the first source operand and the second source operand.

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 ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

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

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.

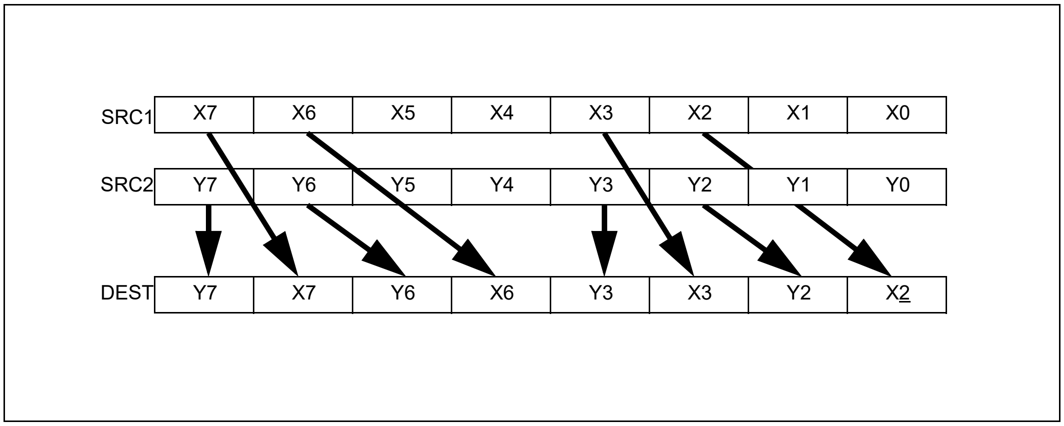


Figure 1-26. VUNPCKHPS Operation

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 using 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 a 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.

Operation

VUNPCKHPS (EVEX Encoded Version When SRC2 is a Register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL >= 128
    TMP_DEST[31:0] := SRC1[95:64]
    TMP_DEST[63:32] := SRC2[95:64]
    TMP_DEST[95:64] := SRC1[127:96]
    TMP_DEST[127:96] := SRC2[127:96]
FI;
IF VL >= 256
    TMP_DEST[159:128] := SRC1[223:192]
    TMP_DEST[191:160] := SRC2[223:192]
    TMP_DEST[223:192] := SRC1[255:224]
    TMP_DEST[255:224] := SRC2[255:224]
FI;
IF VL >= 512
    TMP_DEST[287:256] := SRC1[351:320]
    TMP_DEST[319:288] := SRC2[351:320]
    TMP_DEST[351:320] := SRC1[383:352]
    TMP_DEST[383:352] := SRC2[383:352]
    TMP_DEST[415:384] := SRC1[479:448]
    TMP_DEST[447:416] := SRC2[479:448]
    TMP_DEST[479:448] := SRC1[511:480]
    TMP_DEST[511:480] := SRC2[511:480]
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*                    ; 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
```

VUNPCKHPS (EVEX Encoded Version When SRC2 is Memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[31:0] := SRC1[95:64]
    TMP_DEST[63:32] := TMP_SRC2[95:64]
    TMP_DEST[95:64] := SRC1[127:96]
    TMP_DEST[127:96] := TMP_SRC2[127:96]
FI;
IF VL >= 256
    TMP_DEST[159:128] := SRC1[223:192]
    TMP_DEST[191:160] := TMP_SRC2[223:192]
    TMP_DEST[223:192] := SRC1[255:224]
    TMP_DEST[255:224] := TMP_SRC2[255:224]
FI;
IF VL >= 512
    TMP_DEST[287:256] := SRC1[351:320]
    TMP_DEST[319:288] := TMP_SRC2[351:320]
    TMP_DEST[351:320] := SRC1[383:352]
    TMP_DEST[383:352] := TMP_SRC2[383:352]
    TMP_DEST[415:384] := SRC1[479:448]
    TMP_DEST[447:416] := TMP_SRC2[479:448]
    TMP_DEST[479:448] := SRC1[511:480]
    TMP_DEST[511:480] := TMP_SRC2[511:480]
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*                    ; 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
```

VUNPCKHPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[127:96] := SRC2[127:96]
DEST[159:128] := SRC1[223:192]
DEST[191:160] := SRC2[223:192]
DEST[223:192] := SRC1[255:224]
DEST[255:224] := SRC2[255:224]
DEST[MAXVL-1:256] := 0
```

VUNPCKHPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[127:96] := SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

UNPCKHPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[127:96] := SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VUNPCKHPS __m512 _mm512_unpackhi_ps( __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_mask_unpackhi_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_maskz_unpackhi_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m256 _mm256_unpackhi_ps (__m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_mask_unpackhi_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_maskz_unpackhi_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKHPS __m128 _mm_unpackhi_ps (__m128 a, __m128 b);
VUNPCKHPS __m128 _mm_mask_unpackhi_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKHPS __m128 _mm_maskz_unpackhi_ps(__mmask8 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-50, "Type E4NF Class Exception Conditions."

UNPCKLPD—Unpack and Interleave Low Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 14 /r UNPCKLPD xmm1, xmm2/m128	A	V/V	SSE2	Unpacks and Interleaves double precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.128.66.0F.WIG 14 /r VUNPCKLPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.256.66.0F.WIG 14 /r VUNPCKLPD ymm1,ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.128.66.0F.W1 14 /r VUNPCKLPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Unpacks and Interleaves double precision floating-point values from low quadwords of xmm2 and xmm3/m128/m64bcst subject to write mask k1.
EVEX.256.66.0F.W1 14 /r VUNPCKLPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Unpacks and Interleaves double precision floating-point values from low quadwords of ymm2 and ymm3/m256/m64bcst subject to write mask k1.
EVEX.512.66.0F.W1 14 /r VUNPCKLPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1[1]	Unpacks and Interleaves double precision floating-point values from low quadwords of zmm2 and zmm3/m512/m64bcst subject to write mask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs an interleaved unpack of the low double precision floating-point values from the first source operand and the second source operand.

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 ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

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

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

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 64-bit memory location. The destination operand is a ZMM register, conditionally updated using 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 64-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 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKLPD (EVEX Encoded Versions When SRC2 is a Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL >= 128
    TMP_DEST[63:0] := SRC1[63:0]
    TMP_DEST[127:64] := SRC2[63:0]
FI;
IF VL >= 256
    TMP_DEST[191:128] := SRC1[191:128]
    TMP_DEST[255:192] := SRC2[191:128]
FI;
IF VL >= 512
    TMP_DEST[319:256] := SRC1[319:256]
    TMP_DEST[383:320] := SRC2[319:256]
    TMP_DEST[447:384] := SRC1[447:384]
    TMP_DEST[511:448] := SRC2[447:384]
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
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKLPD (EVEX Encoded Version When SRC2 is Memory)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[63:0] := SRC1[63:0]
    TMP_DEST[127:64] := TMP_SRC2[63:0]
FI;
IF VL >= 256
    TMP_DEST[191:128] := SRC1[191:128]
    TMP_DEST[255:192] := TMP_SRC2[191:128]
FI;
IF VL >= 512
    TMP_DEST[319:256] := SRC1[319:256]
    TMP_DEST[383:320] := TMP_SRC2[319:256]
    TMP_DEST[447:384] := SRC1[447:384]
    TMP_DEST[511:448] := TMP_SRC2[447:384]
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
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKLPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[191:128] := SRC1[191:128]
DEST[255:192] := SRC2[191:128]
DEST[MAXVL-1:256] := 0
```

VUNPCKLPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0
```

UNPCKLPD (128-bit Legacy SSE Version)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKLPD __m512d _mm512_unpacklo_pd(__m512d a, __m512d b);
VUNPCKLPD __m512d _mm512_mask_unpacklo_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKLPD __m512d _mm512_maskz_unpacklo_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKLPD __m256d _mm256_unpacklo_pd(__m256d a, __m256d b)
VUNPCKLPD __m256d _mm256_mask_unpacklo_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKLPD __m256d _mm256_maskz_unpacklo_pd(__mmask8 k, __m256d a, __m256d b);
UNPCKLPD __m128d _mm_unpacklo_pd(__m128d a, __m128d b)
VUNPCKLPD __m128d _mm_mask_unpacklo_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKLPD __m128d _mm_maskz_unpacklo_pd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-50, “Type E4NF Class Exception Conditions.”

UNPCKLPS—Unpack and Interleave Low Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 14 /r UNPCKLPS xmm1, xmm2/m128	A	V/V	SSE	Unpacks and Interleaves single precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.128.0F.WIG 14 /r VUNPCKLPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves single precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.256.0F.WIG 14 /r VUNPCKLPS ymm1,ymm2,ymm3/m256	B	V/V	AVX	Unpacks and Interleaves single precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.128.0F.W0 14 /r VUNPCKLPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Unpacks and Interleaves single precision floating-point values from low quadwords of xmm2 and xmm3/mem and write result to xmm1 subject to write mask k1.
EVEX.256.0F.W0 14 /r VUNPCKLPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Unpacks and Interleaves single precision floating-point values from low quadwords of ymm2 and ymm3/mem and write result to ymm1 subject to write mask k1.
EVEX.512.0F.W0 14 /r VUNPCKLPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1[1]	Unpacks and Interleaves single precision floating-point values from low quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to write mask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

Performs an interleaved unpack of the low single precision floating-point values from the first source operand and the second source operand.

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 ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

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

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

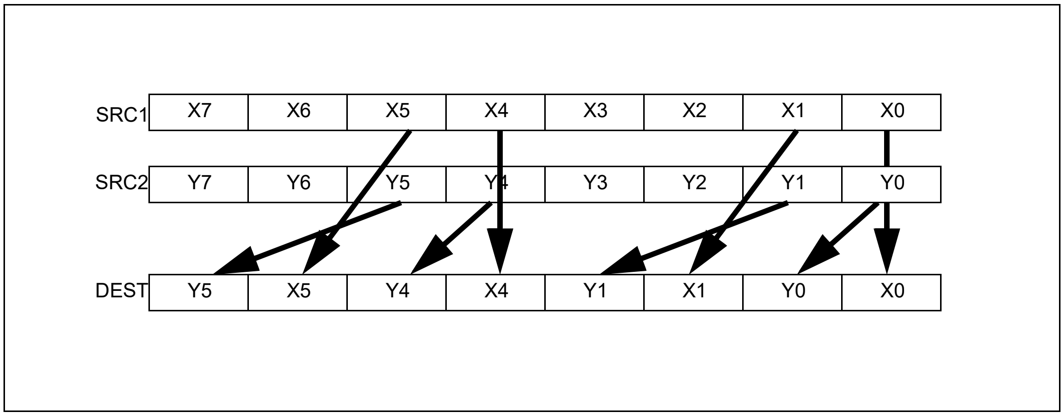


Figure 1-27. VUNPCKLPS Operation

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 using 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.

Operation

VUNPCKLPS (EVEX Encoded Version When SRC2 is a ZMM Register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL >= 128
    TMP_DEST[31:0] := SRC1[31:0]
    TMP_DEST[63:32] := SRC2[31:0]
    TMP_DEST[95:64] := SRC1[63:32]
    TMP_DEST[127:96] := SRC2[63:32]
FI;
IF VL >= 256
    TMP_DEST[159:128] := SRC1[159:128]
    TMP_DEST[191:160] := SRC2[159:128]
    TMP_DEST[223:192] := SRC1[191:160]
    TMP_DEST[255:224] := SRC2[191:160]
FI;
IF VL >= 512
    TMP_DEST[287:256] := SRC1[287:256]
    TMP_DEST[319:288] := SRC2[287:256]
    TMP_DEST[351:320] := SRC1[319:288]
    TMP_DEST[383:352] := SRC2[319:288]
    TMP_DEST[415:384] := SRC1[415:384]
    TMP_DEST[447:416] := SRC2[415:384]
    TMP_DEST[479:448] := SRC1[447:416]
    TMP_DEST[511:480] := SRC2[447:416]
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
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKLPS (EVEX Encoded Version When SRC2 is Memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 31
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
TMP_DEST[31:0] := SRC1[31:0]
TMP_DEST[63:32] := TMP_SRC2[31:0]
TMP_DEST[95:64] := SRC1[63:32]
TMP_DEST[127:96] := TMP_SRC2[63:32]
FI;
IF VL >= 256
    TMP_DEST[159:128] := SRC1[159:128]
    TMP_DEST[191:160] := TMP_SRC2[159:128]
    TMP_DEST[223:192] := SRC1[191:160]
    TMP_DEST[255:224] := TMP_SRC2[191:160]
FI;
IF VL >= 512
    TMP_DEST[287:256] := SRC1[287:256]
    TMP_DEST[319:288] := TMP_SRC2[287:256]
    TMP_DEST[351:320] := SRC1[319:288]
    TMP_DEST[383:352] := TMP_SRC2[319:288]
    TMP_DEST[415:384] := SRC1[415:384]
    TMP_DEST[447:416] := TMP_SRC2[415:384]
    TMP_DEST[479:448] := SRC1[447:416]
    TMP_DEST[511:480] := TMP_SRC2[447:416]
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
                    DEST[i+31:i] := 0
            FI
```

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

UNPCKLPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[127:96] := SRC2[63:32]
DEST[159:128] := SRC1[159:128]
DEST[191:160] := SRC2[159:128]
DEST[223:192] := SRC1[191:160]
DEST[255:224] := SRC2[191:160]
DEST[MAXVL-1:256] := 0
```

VUNPCKLPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[127:96] := SRC2[63:32]
DEST[MAXVL-1:128] := 0
```

UNPCKLPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[127:96] := SRC2[63:32]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VUNPCKLPS __m512 _mm512_unpacklo_ps(__m512 a, __m512 b);
VUNPCKLPS __m512 _mm512_mask_unpacklo_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m512 _mm512_maskz_unpacklo_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m256 _mm256_unpacklo_ps (__m256 a, __m256 b);
VUNPCKLPS __m256 _mm256_mask_unpacklo_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKLPS __m256 _mm256_maskz_unpacklo_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKLPS __m128 _mm_unpacklo_ps (__m128 a, __m128 b);
VUNPCKLPS __m128 _mm_mask_unpacklo_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKLPS __m128 _mm_maskz_unpacklo_ps(__mmask8 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-50, “Type E4NF Class Exception Conditions.”

VADDPH—Add Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 58 /r VADDPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Add packed FP16 value from xmm3/m128/m16bcst to xmm2, and store result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 58 /r VADDPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Add packed FP16 value from ymm3/m256/m16bcst to ymm2, and store result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 58 /r VADDPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Add packed FP16 value from zmm3/m512/m16bcst to zmm2, and store result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

Operation

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

```
VL = 128, 256 or 512
KL := VL/16
IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[j]
    ELSEIF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
DEST[MAXVL-1:VL] := 0
```

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

```
VL = 128, 256 or 512
KL := VL/16
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[0]
        ELSE:
            DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VADDPH __m128h _mm_add_ph (__m128h a, __m128h b);
VADDPH __m128h _mm_mask_add_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VADDPH __m128h _mm_maskz_add_ph (__mmask8 k, __m128h a, __m128h b);
VADDPH __m256h _mm256_add_ph (__m256h a, __m256h b);
VADDPH __m256h _mm256_mask_add_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VADDPH __m256h _mm256_maskz_add_ph (__mmask16 k, __m256h a, __m256h b);
VADDPH __m512h _mm512_add_ph (__m512h a, __m512h b);
VADDPH __m512h _mm512_add_ph (__m512h a, __m512h b);
VADDPH __m512h _mm512_mask_add_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VADDPH __m512h _mm512_maskz_add_ph (__mmask32 k, __m512h a, __m512h b);
VADDPH __m512h _mm512_add_round_ph (__m512h a, __m512h b, int rounding);
VADDPH __m512h _mm512_mask_add_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VADDPH __m512h _mm512_maskz_add_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

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

VADDSH—Add Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 58 /r VADDSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Add the low FP16 value from xmm3/m16 to xmm2, and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

Operation

VADDSH (EVEX Encoded Versions)

```
IF EVEX.b = 1 and SRC2 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)
IF k1[0] OR *no writemask*:
    DEST.fp16[0] := SRC1.fp16[0] + SRC2.fp16[0]
ELSEIF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged
DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VADDSH __m128h _mm_add_round_sh (__m128h a, __m128h b, int rounding);
VADDSH ___m128h _mm_mask_add_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
VADDSH ___m128h _mm_maskz_add_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
VADDSH ___m128h _mm_add_sh (__m128h a, __m128h b);
VADDSH ___m128h _mm_mask_add_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VADDSH ___m128h _mm_maskz_add_sh (__mmask8 k, __m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

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

VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

Operation

VALIGND (EVEX Encoded Versions)

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

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

VALIGNQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256),(8, 512)
IF (SRC2 *is memory*) (AND EVEX.b = 1)
    THEN
        FOR j := 0 TO KL-1
            i := j * 64
            src[i+63:i] := SRC2[63:0]
        ENDFOR;
    ELSE src := SRC2
FI
; Concatenate sources
tmp[VL-1:0] := src[VL-1:0]
tmp[2VL-1:VL] := SRC1[VL-1:0]
; Shift right quadword elements
```

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VALIGND __m512i _mm512_alignr_epi32( __m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_mask_alignr_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_maskz_alignr_epi32( __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m256i _mm256_mask_alignr_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m256i _mm256_maskz_alignr_epi32( __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m128i _mm_mask_alignr_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGND __m128i _mm_maskz_alignr_epi32( __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m512i _mm512_alignr_epi64( __m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_mask_alignr_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_maskz_alignr_epi64( __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m256i _mm256_mask_alignr_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m256i _mm256_maskz_alignr_epi64( __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m128i _mm_mask_alignr_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m128i _mm_maskz_alignr_epi64( __mmask8 k, __m128i a, __m128i b, int cnt);
```

Exceptions

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

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

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Blend single precision vector zmm2 and single precision vector zmm3/m512/m32bcst using k1 as select control and store the result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

VBLENDMPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := SRC2[63:0]
                ELSE
                    DEST[i+63:i] := SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN DEST[i+63:i] := SRC1[i+63:i]
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBLENDMPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN DEST[i+31:i] := SRC1[i+31:i]
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

None.

Other Exceptions

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

VBROADCAST—Load with Broadcast Floating-Point Data

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	Broadcast 256 bits of 4 double precision floating-point data in mem to locations in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2/VBROADCASTF32X8/VBROADCASTF64X4 load floating-point values as tuples from the source operand (the second operand) in memory or register and broadcast to all elements of the destination operand (the first operand). The destination operand is a YMM/ZMM register updated according to the writemask k1. The source operand is either a register or 64-bit/128-bit/256-bit memory location.

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

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF32X8 have 32-bit granularity. VBROADCASTF64X2 and VBROADCASTF64X4 have 64-bit granularity.

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

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

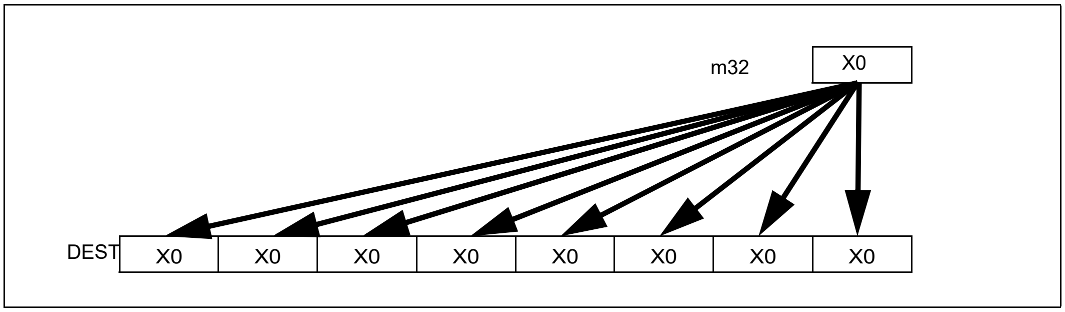


Figure 1-28. VBROADCASTSS Operation (VEX.256 encoded version)

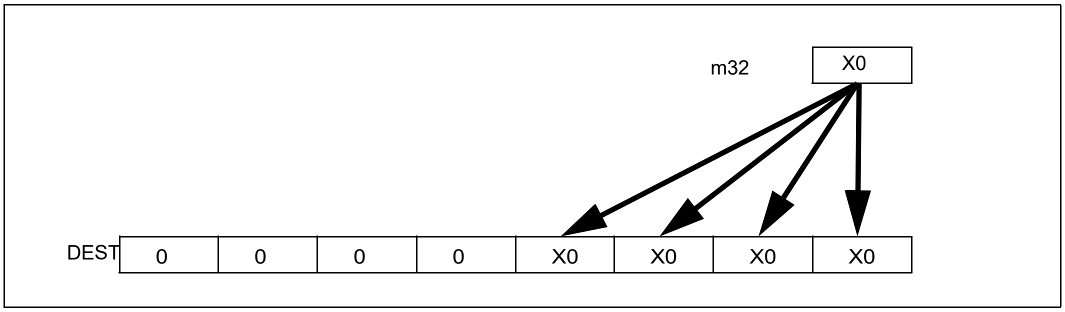


Figure 1-29. VBROADCASTSS Operation (VEX.128-bit version)

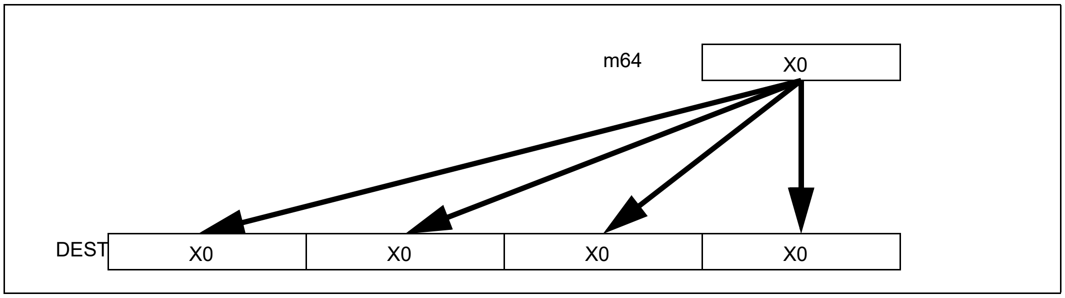


Figure 1-30. VBROADCASTSD Operation (VEX.256-bit version)

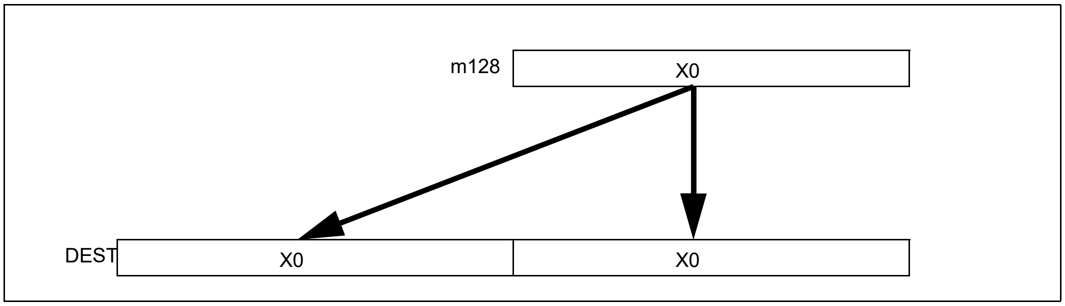


Figure 1-31. VBROADCASTF128 Operation (VEX.256-bit version)

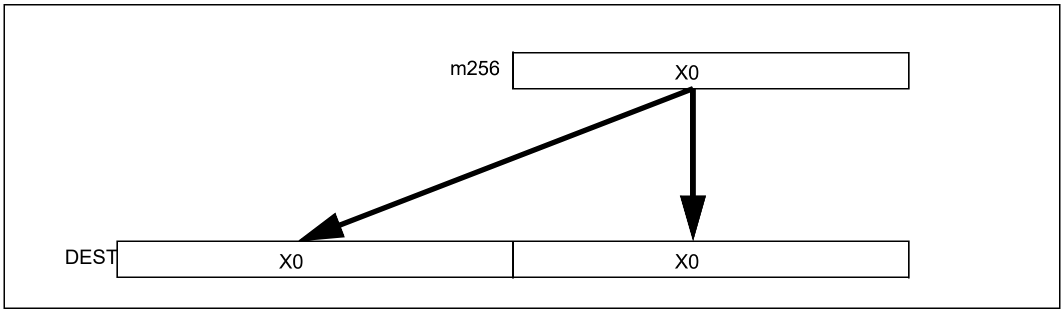


Figure 1-32. VBROADCASTF64X4 Operation (512-bit version with writemask all 1s)

Operation

VBROADCASTSS (128-bit Version VEX and Legacy)

```
temp := SRC[31:0]
DEST[31:0] := temp
DEST[63:32] := temp
DEST[95:64] := temp
DEST[127:96] := temp
DEST[MAXVL-1:128] := 0
```

VBROADCASTSS (VEX.256 Encoded Version)

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

VBROADCASTSS (EVEX Encoded Versions)

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

VBROADCASTSD (VEX.256 Encoded Version)

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

VBROADCASTSD (EVEX Encoded Versions)

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

VBROADCASTF32x2 (EVEX Encoded Versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    n := (j mod 2) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBROADCASTF128 (VEX.256 Encoded Version)

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

VBROADCASTF32X4 (EVEX Encoded Versions)

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

VBROADCASTF64X2 (EVEX Encoded Versions)

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

VBROADCASTF32X8 (EVEX.U1.512 Encoded Version)

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

VBROADCASTF64X4 (EVEX.512 Encoded Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VBROADCASTF32x2 __m512 _mm512_broadcast_f32x2(__m128 a);
VBROADCASTF32x2 __m512 _mm512_mask_broadcast_f32x2(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x2 __m512 _mm512_maskz_broadcast_f32x2(__mmask16 k, __m128 a);
VBROADCASTF32x2 __m256 _mm256_broadcast_f32x2(__m128 a);
VBROADCASTF32x2 __m256 _mm256_mask_broadcast_f32x2(__m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x2 __m256 _mm256_maskz_broadcast_f32x2(__mmask8 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_broadcast_f32x4(__m128 a);
VBROADCASTF32x4 __m512 _mm512_mask_broadcast_f32x4(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_maskz_broadcast_f32x4(__mmask16 k, __m128 a);
VBROADCASTF32x4 __m256 _mm256_broadcast_f32x4(__m128 a);
VBROADCASTF32x4 __m256 _mm256_mask_broadcast_f32x4(__m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x4 __m256 _mm256_maskz_broadcast_f32x4(__mmask8 k, __m128 a);
VBROADCASTF32x8 __m512 _mm512_broadcast_f32x8(__m256 a);
VBROADCASTF32x8 __m512 _mm512_mask_broadcast_f32x8(__m512 s, __mmask16 k, __m256 a);
VBROADCASTF32x8 __m512 _mm512_maskz_broadcast_f32x8(__mmask16 k, __m256 a);
VBROADCASTF64x2 __m512d _mm512_broadcast_f64x2(__m128d a);
VBROADCASTF64x2 __m512d _mm512_mask_broadcast_f64x2(__m512d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m512d _mm512_maskz_broadcast_f64x2(__mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256_broadcast_f64x2(__m128d a);
VBROADCASTF64x2 __m256d _mm256_mask_broadcast_f64x2(__m256d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256_maskz_broadcast_f64x2(__mmask8 k, __m128d a);
VBROADCASTF64x4 __m512d _mm512_broadcast_f64x4(__m256d a);
VBROADCASTF64x4 __m512d _mm512_mask_broadcast_f64x4(__m512d s, __mmask8 k, __m256d a);
VBROADCASTF64x4 __m512d _mm512_maskz_broadcast_f64x4(__mmask8 k, __m256d a);
VBROADCASTSD __m512d _mm512_broadcastsd_pd(__m128d a);
VBROADCASTSD __m512d _mm512_mask_broadcastsd_pd(__m512d s, __mmask8 k, __m128d a);
VBROADCASTSD __m512d _mm512_maskz_broadcastsd_pd(__mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_broadcastsd_pd(__m128d a);
VBROADCASTSD __m256d _mm256_mask_broadcastsd_pd(__m256d s, __mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_maskz_broadcastsd_pd(__mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_broadcast_sd(double *a);
VBROADCASTSS __m512 _mm512_broadcastss_ps(__m128 a);
VBROADCASTSS __m512 _mm512_mask_broadcastss_ps(__m512 s, __mmask16 k, __m128 a);
VBROADCASTSS __m512 _mm512_maskz_broadcastss_ps(__mmask16 k, __m128 a);
VBROADCASTSS __m256 _mm256_broadcastss_ps(__m128 a);
VBROADCASTSS __m256 _mm256_mask_broadcastss_ps(__m256 s, __mmask8 k, __m128 a);
VBROADCASTSS __m256 _mm256_maskz_broadcastss_ps(__mmask8 k, __m128 a);

```
VBROADCASTSS __m128 _mm_broadcastss_ps(__m128 a);
VBROADCASTSS __m128 _mm_mask_broadcastss_ps(__m128 s, __mmask8 k, __m128 a);
VBROADCASTSS __m128 _mm_maskz_broadcastss_ps( __mmask8 k, __m128 a);
VBROADCASTSS __m128 _mm_broadcast_ss(float *a);
VBROADCASTSS __m256 _mm256_broadcast_ss(float *a);
VBROADCASTF128 __m256 _mm256_broadcast_ps(__m128 * a);
VBROADCASTF128 __m256d _mm256_broadcast_pd(__m128d * a);
```

Exceptions

VEX-encoded instructions, see Table 2-23, “Type 6 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-53, “Type E6 Class Exception Conditions.”

Additionally:

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

VCMPPH—Compare Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.0F3A.W0 C2 /r /ib VCMPPH k1{k2}, xmm2, xmm3/m128/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Compare packed FP16 values in xmm3/m128/m16bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.
EVEX.256.NP.0F3A.W0 C2 /r /ib VCMPPH k1{k2}, ymm2, ymm3/m256/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Compare packed FP16 values in ymm3/m256/m16bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.
EVEX.512.NP.0F3A.W0 C2 /r /ib VCMPPH k1{k2}, zmm2, zmm3/m512/m16bcst {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1[1]	Compare packed FP16 values in zmm3/m512/m16bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

Operation

```
CASE (imm8 & 0x1F) OF
0: CMP_OPERATOR := EQ_OQ;
1: CMP_OPERATOR := LT_OS;
2: CMP_OPERATOR := LE_OS;
3: CMP_OPERATOR := UNORD_Q;
4: CMP_OPERATOR := NEQ_UQ;
5: CMP_OPERATOR := NLT_US;
6: CMP_OPERATOR := NLE_US;
7: CMP_OPERATOR := ORD_Q;
8: CMP_OPERATOR := EQ_UQ;
9: CMP_OPERATOR := NGE_US;
10: CMP_OPERATOR := NGT_US;
11: CMP_OPERATOR := FALSE_OQ;
12: CMP_OPERATOR := NEQ_OQ;
13: CMP_OPERATOR := GE_OS;
14: CMP_OPERATOR := GT_OS;
15: CMP_OPERATOR := TRUE_UQ;
16: CMP_OPERATOR := EQ_OS;
```

```
17: CMP_OPERATOR := LT_OQ;
18: CMP_OPERATOR := LE_OQ;
19: CMP_OPERATOR := UNORD_S;
20: CMP_OPERATOR := NEQ_US;
21: CMP_OPERATOR := NLT_UQ;
22: CMP_OPERATOR := NLE_UQ;
23: CMP_OPERATOR := ORD_S;
24: CMP_OPERATOR := EQ_US;
25: CMP_OPERATOR := NGE_UQ;
26: CMP_OPERATOR := NGT_UQ;
27: CMP_OPERATOR := FALSE_OS;
28: CMP_OPERATOR := NEQ_OS;
29: CMP_OPERATOR := GE_OQ;
30: CMP_OPERATOR := GT_OQ;
31: CMP_OPERATOR := TRUE_US;
ESAC
```

VCMPPH (EVEX Encoded Versions)

```
VL = 128, 256 or 512
KL := VL/16

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

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

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

VCMPSH—Compare Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.0F3A.W0 C2 /r /ib VCMPSH k1{k2}, xmm2, xmm3/m16 {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1[1]	Compare low FP16 values in xmm3/m16 and xmm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

Operation

```
CASE (imm8 & 0x1F) OF
0: CMP_OPERATOR := EQ_OQ;
1: CMP_OPERATOR := LT_OS;
2: CMP_OPERATOR := LE_OS;
3: CMP_OPERATOR := UNORD_Q;
4: CMP_OPERATOR := NEQ_UQ;
5: CMP_OPERATOR := NLT_US;
6: CMP_OPERATOR := NLE_US;
7: CMP_OPERATOR := ORD_Q;
8: CMP_OPERATOR := EQ_UQ;
9: CMP_OPERATOR := NGE_US;
10: CMP_OPERATOR := NGT_US;
11: CMP_OPERATOR := FALSE_OQ;
12: CMP_OPERATOR := NEQ_OQ;
13: CMP_OPERATOR := GE_OS;
14: CMP_OPERATOR := GT_OS;
15: CMP_OPERATOR := TRUE_UQ;
16: CMP_OPERATOR := EQ_OS;
17: CMP_OPERATOR := LT_OQ;
18: CMP_OPERATOR := LE_OQ;
19: CMP_OPERATOR := UNORD_S;
20: CMP_OPERATOR := NEQ_US;
21: CMP_OPERATOR := NLT_UQ;
22: CMP_OPERATOR := NLE_UQ;
23: CMP_OPERATOR := ORD_S;
24: CMP_OPERATOR := EQ_US;
25: CMP_OPERATOR := NGE_UQ;
```

```
26: CMP_OPERATOR := NGT_UQ;
27: CMP_OPERATOR := FALSE_OS;
28: CMP_OPERATOR := NEQ_OS;
29: CMP_OPERATOR := GE_OQ;
30: CMP_OPERATOR := GT_OQ;
31: CMP_OPERATOR := TRUE_US;
ESAC
```

VCMPSH (EVEX Encoded Versions)

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

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

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

VCOMISH—Compare Scalar Ordered FP16 Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.MAP5.W0 2F /r VCOMISH xmm1, xmm2/m16 {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Compare low FP16 values in xmm1 and xmm2/m16, and set the EFLAGS flags accordingly.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

VCOMISH SRC1, SRC2

```
RESULT := OrderedCompare(SRC1.fp16[0],SRC2.fp16[0])
IF RESULT is UNORDERED:
    ZF, PF, CF := 1, 1, 1
ELSE IF RESULT is GREATER_THAN:
    ZF, PF, CF := 0, 0, 0
ELSE IF RESULT is LESS_THAN:
    ZF, PF, CF := 0, 0, 1
ELSE: // RESULT is EQUALS
    ZF, PF, CF := 1, 0, 0

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VCOMISH int _mm_comi_round_sh (__m128h a, __m128h b, const int imm8, const int sae);
VCOMISH int _mm_comi_sh (__m128h a, __m128h b, const int imm8);
VCOMISH int _mm_comieq_sh (__m128h a, __m128h b);
VCOMISH int _mm_comige_sh (__m128h a, __m128h b);
VCOMISH int _mm_comigt_sh (__m128h a, __m128h b);
VCOMISH int _mm_comile_sh (__m128h a, __m128h b);
VCOMISH int _mm_comilt_sh (__m128h a, __m128h b);
VCOMISH int _mm_comineq_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

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

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

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Compress packed double precision floating-point values from zmm2 using control mask k1 to zmm1/m512.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

Operation

VCOMPRESSPD (EVEX Encoded Versions) Store Form

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

```
    FI;
ENDFOR
```

VCOMPRESSPD (EVEX Encoded Versions) Reg-Reg Form

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

None.

Other Exceptions

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

Additionally:

#UD	If EVEX.vvvv != 1111B.

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

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Compress packed single precision floating-point values from zmm2 using control mask k1 to zmm1/m512.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

Operation

VCOMPRESSPS (EVEX Encoded Versions) Store Form

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE := 32
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+31:i]
            k := k + SIZE
    FI;
```

```
ENDFOR;
```

VCOMPRESSPS (EVEX Encoded Versions) Reg-Reg Form

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VCOMPRESSPS __m512 _mm512_mask_compress_ps( __m512 s, __mmask16 k, __m512 a);
VCOMPRESSPS __m512 _mm512_maskz_compress_ps( __mmask16 k, __m512 a);
VCOMPRESSPS void _mm512_mask_compressstoreu_ps( void * d, __mmask16 k, __m512 a);
VCOMPRESSPS __m256 _mm256_mask_compress_ps( __m256 s, __mmask8 k, __m256 a);
VCOMPRESSPS __m256 _mm256_maskz_compress_ps( __mmask8 k, __m256 a);
VCOMPRESSPS void _mm256_mask_compressstoreu_ps( void * d, __mmask8 k, __m256 a);
VCOMPRESSPS __m128 _mm_mask_compress_ps( __m128 s, __mmask8 k, __m128 a);
VCOMPRESSPS __m128 _mm_maskz_compress_ps( __mmask8 k, __m128 a);
VCOMPRESSPS void _mm_mask_compressstoreu_ps( void * d, __mmask8 k, __m128 a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

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

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTDQ2PH—Convert Packed Signed Doubleword Integers to Packed FP16 Values

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EVEX.128.NP.MAP5.W0 5B /r VCVTDQ2PH xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5B /r VCVTDQ2PH xmm1{k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5B /r VCVTDQ2PH ymm1{k1}{z}, zmm2/m512/m32bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed FP16 values, and store the result in ymm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

VCVTDQ2PH DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 32

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.dword[0]
        ELSE
            tsrc := SRC.dword[j]
        DEST.fp16[j] := Convert_integer32_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTDQ2PH __m256h _mm512_cvt_roundepi32_ph (__m512i a, int rounding);
VCVTDQ2PH __m256h _mm512_mask_cvt_roundepi32_ph (__m256h src, __mmask16 k, __m512i a, int rounding);
VCVTDQ2PH __m256h _mm512_maskz_cvt_roundepi32_ph (__mmask16 k, __m512i a, int rounding);
VCVTDQ2PH __m128h _mm_cvtepi32_ph (__m128i a);
VCVTDQ2PH __m128h _mm_mask_cvtepi32_ph (__m128h src, __mmask8 k, __m128i a);
VCVTDQ2PH __m128h _mm_maskz_cvtepi32_ph (__mmask8 k, __m128i a);
VCVTDQ2PH __m128h _mm256_cvtepi32_ph (__m256i a);
VCVTDQ2PH __m128h _mm256_mask_cvtepi32_ph (__m128h src, __mmask8 k, __m256i a);
VCVTDQ2PH __m128h _mm256_maskz_cvtepi32_ph (__mmask8 k, __m256i a);
VCVTDQ2PH __m256h _mm512_cvtepi32_ph (__m512i a);
VCVTDQ2PH __m256h _mm512_mask_cvtepi32_ph (__m256h src, __mmask16 k, __m512i a);
VCVTDQ2PH __m256h _mm512_maskz_cvtepi32_ph (__mmask16 k, __m512i a);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

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

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F38.W0 72 /r VCVTNE2PS2BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512_BF16 AND AVX512VL) OR AVX10.1[1]	Convert packed single data from xmm2 and xmm3/m128/m32bcst to packed BF16 data in xmm1 with writemask k1.
EVEX.256.F2.0F38.W0 72 /r VCVTNE2PS2BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512_BF16 AND AVX512VL) OR AVX10.1[1]	Convert packed single data from ymm2 and ymm3/m256/m32bcst to packed BF16 data in ymm1 with writemask k1.
EVEX.512.F2.0F38.W0 72 /r VCVTNE2PS2BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	(AVX512_BF16 AND AVX512F) OR AVX10.1[1]	Convert packed single data from zmm2 and zmm3/m512/m32bcst to packed BF16 data in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

This instruction does not support memory fault suppression.

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

Operation

VCVTNE2PS2BF16 dest, src1, src2

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

origdest := dest
FOR i := 0 to KL-1:
    IF k1[ i ] or *no writemask*:
        IF i < KL/2:
            IF src2 is memory and evex.b == 1:
                t := src2.fp32[0]
            ELSE:
                t := src2.fp32[ i ]
        ELSE:
            t := src1.fp32[ i-KL/2]

        // See VCVTNEPS2BF16 for definition of convert helper function
        dest.word[i] := convert_fp32_to_bfloat16(t)

    ELSE IF *zeroing*:
        dest.word[ i ] := 0
    ELSE:  // Merge masking, dest element unchanged
```

```
        dest.word[ i ] := origdest.word[ i ]
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTNE2PS2BF16 __m128bh _mm_cvtne2ps_pbh (__m128, __m128);
VCVTNE2PS2BF16 __m128bh _mm_mask_cvtne2ps_pbh (__m128bh, __mmask8, __m128, __m128);
VCVTNE2PS2BF16 __m128bh _mm_maskz_cvtne2ps_pbh (__mmask8, __m128, __m128);
VCVTNE2PS2BF16 __m256bh _mm256_cvtne2ps_pbh (__m256, __m256);
VCVTNE2PS2BF16 __m256bh _mm256_mask_cvtne2ps_pbh (__m256bh, __mmask16, __m256, __m256);
VCVTNE2PS2BF16 __m256bh _mm256_maskz_cvtne2ps_ pbh (__mmask16, __m256, __m256);
VCVTNE2PS2BF16 __m512bh _mm512_cvtne2ps_pbh (__m512, __m512);
VCVTNE2PS2BF16 __m512bh _mm512_mask_cvtne2ps_pbh (__m512bh, __mmask32, __m512, __m512);
VCVTNE2PS2BF16 __m512bh _mm512_maskz_cvtne2ps_pbh (__mmask32, __m512, __m512);

SIMD Floating-Point Exceptions

None.

Other Exceptions

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

VCVTNEPS2BF16—Convert Packed Single Data to Packed BF16 Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512_BF16 AND AVX512VL) OR AVX10.1[1]	Convert packed single data from xmm2/m128 to packed BF16 data in xmm1 with writemask k1.
EVEX.256.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1{k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512_BF16 AND AVX512VL) OR AVX10.1[1]	Convert packed single data from ymm2/m256 to packed BF16 data in xmm1 with writemask k1.
EVEX.512.F3.0F38.W0 72 /r VCVTNEPS2BF16 ymm1{k1}{z}, zmm2/m512/m32bcst	A	V/V	(AVX512_BF16 AND AVX512F) OR AVX10.1[1]	Convert packed single data from zmm2/m512 to packed BF16 data in ymm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

Operation

```
Define convert_fp32_to_bfloat16(x):
    IF x is zero or denormal:
        dest[15] := x[31] // sign preserving zero (denormal go to zero)
        dest[14:0] := 0
    ELSE IF x is infinity:
        dest[15:0] := x[31:16]
    ELSE IF x is NAN:
        dest[15:0] := x[31:16] // truncate and set MSB of the mantissa to force QNAN
        dest[6] := 1
    ELSE // normal number
        LSB := x[16]
        rounding_bias := 0x00007FFF + LSB
        temp[31:0] := x[31:0] + rounding_bias // integer add
        dest[15:0] := temp[31:16]
    RETURN dest
```

```
VCVTNEPS2BF16 dest, src
VL = (128, 256, 512)
KL = VL/16

origdest := dest
FOR i := 0 to KL/2-1:
    IF k1[ i ] or *no writemask*:
        IF src is memory and evex.b == 1:
            t := src.fp32[0]
        ELSE:
            t := src.fp32[ i ]

        dest.word[i] := convert_fp32_to_bfloat16(t)

    ELSE IF *zeroing*:
        dest.word[ i ] := 0
    ELSE:  // Merge masking, dest element unchanged
        dest.word[ i ] := origdest.word[ i ]
DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTNEPS2BF16 __m128bh _mm_cvtneps_pbh (__m128);
VCVTNEPS2BF16 __m128bh _mm_mask_cvtneps_pbh (__m128bh, __mmask8, __m128);
VCVTNEPS2BF16 __m128bh _mm_maskz_cvtneps_pbh (__mmask8, __m128);
VCVTNEPS2BF16 __m128bh _mm256_cvtneps_pbh (__m256);
VCVTNEPS2BF16 __m128bh _mm256_mask_cvtneps_pbh (__m128bh, __mmask8, __m256);
VCVTNEPS2BF16 __m128bh _mm256_maskz_cvtneps_pbh (__mmask8, __m256);
VCVTNEPS2BF16 __m256bh _mm512_cvtneps_pbh (__m512);
VCVTNEPS2BF16 __m256bh _mm512_mask_cvtneps_pbh (__m256bh, __mmask16, __m512);
VCVTNEPS2BF16 __m256bh _mm512_maskz_cvtneps_pbh (__mmask16, __m512);

SIMD Floating-Point Exceptions

None.

Other Exceptions

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

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W1 5A /r VCVTPD2PH xmm1{k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP5.W1 5A /r VCVTPD2PH xmm1{k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.66.MAP5.W1 5A /r VCVTPD2PH xmm1{k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight packed FP16 values, and store the result in ymm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

This instruction converts two, four, or eight packed double precision floating-point values in the source operand (second operand) to two, four, or eight packed FP16 values in the destination operand (first operand). When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasts from a 64-bit memory location. The destination operand is a XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:128/64/32) of the corresponding destination are zeroed.

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

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

Operation

```
VCVTPD2PH DEST, SRC
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.double[0]
        ELSE
            tsrc := SRC.double[j]
        DEST.fp16[j] := Convert_fp64_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPD2PH __m128h _mm512_cvt_roundpd_ph (__m512d a, int rounding);
VCVTPD2PH __m128h _mm512_mask_cvt_roundpd_ph (__m128h src, __mmask8 k, __m512d a, int rounding);
VCVTPD2PH __m128h _mm512_maskz_cvt_roundpd_ph (__mmask8 k, __m512d a, int rounding);
VCVTPD2PH __m128h _mm_cvtpd_ph (__m128d a);
VCVTPD2PH __m128h _mm_mask_cvtpd_ph (__m128h src, __mmask8 k, __m128d a);
VCVTPD2PH __m128h _mm_maskz_cvtpd_ph (__mmask8 k, __m128d a);
VCVTPD2PH __m128h _mm256_cvtpd_ph (__m256d a);
VCVTPD2PH __m128h _mm256_mask_cvtpd_ph (__m128h src, __mmask8 k, __m256d a);
VCVTPD2PH __m128h _mm256_maskz_cvtpd_ph (__mmask8 k, __m256d a);
VCVTPD2PH __m128h _mm512_cvtpd_ph (__m512d a);
VCVTPD2PH __m128h _mm512_mask_cvtpd_ph (__m128h src, __mmask8 k, __m512d a);
VCVTPD2PH __m128h _mm512_maskz_cvtpd_ph (__mmask8 k, __m512d a);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

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

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

Opcode/ 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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	Convert eight packed double precision floating-point values from zmm2/m512/m64bcst to eight packed quadword integers in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

VCVTPD2QQ (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

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

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

VCVTPD2QQ (EVEX Encoded Version) When SRC Operand is a Memory Source

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

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTPD2UDQ—Convert Packed Double Precision Floating-Point Values to Packed Unsigned Doubleword Integers

Opcode 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

VCVTPD2UDQ (EVEX Encoded Versions) When SRC2 Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
```

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

VCVTPD2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

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

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPD2UDQ __m256i _mm512_cvtpd_epu32( __m512d a);
VCVTPD2UDQ __m256i _mm512_mask_cvtpd_epu32( __m256i s, __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_maskz_cvtpd_epu32( __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_cvt_roundpd_epu32( __m512d a, int r);
VCVTPD2UDQ __m256i _mm512_mask_cvt_roundpd_epu32( __m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m256i _mm512_maskz_cvt_roundpd_epu32( __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m128i _mm256_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm256_maskz_cvtpd_epu32( __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m128d a);
VCVTPD2UDQ __m128i _mm_maskz_cvtpd_epu32( __mmask8 k, __m128d a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

Additionally:

#UD	If EVEX.vvvv != 1111B.

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

Opcode/ 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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	Convert eight packed double precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

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

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

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

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

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

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

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPD2UQQ __m512i _mm512_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 Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTPH2DQ—Convert Packed FP16 Values to Signed Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 5B /r VCVTPH2DQ xmm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert four packed FP16 values in xmm2/m64/m16bcst to four signed doubleword integers, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP5.W0 5B /r VCVTPH2DQ ymm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed doubleword integers, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP5.W0 5B /r VCVTPH2DQ zmm1{k1}{z}, ymm2/m256/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen signed doubleword integers, and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

The destination elements are updated according to the writemask.

Operation

VCVTPH2DQ DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 32

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.dword[j] := Convert_fp16_to_integer32(tsrc)
```

```
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    // else dest.dword[j] remains unchanged

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2DQ __m512i _mm512_cvt_roundph_epi32 (__m256h a, int rounding);
VCVTPH2DQ __m512i _mm512_mask_cvt_roundph_epi32 (__m512i src, __mmask16 k, __m256h a, int rounding);
VCVTPH2DQ __m512i _mm512_maskz_cvt_roundph_epi32 (__mmask16 k, __m256h a, int rounding);
VCVTPH2DQ __m128i _mm_cvtph_epi32 (__m128h a);
VCVTPH2DQ __m128i _mm_mask_cvtph_epi32 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2DQ __m128i _mm_maskz_cvtph_epi32 (__mmask8 k, __m128h a);
VCVTPH2DQ __m256i _mm256_cvtph_epi32 (__m128h a);
VCVTPH2DQ __m256i _mm256_mask_cvtph_epi32 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2DQ __m256i _mm256_maskz_cvtph_epi32 (__mmask8 k, __m128h a);
VCVTPH2DQ __m512i _mm512_cvtph_epi32 (__m256h a);
VCVTPH2DQ __m512i _mm512_mask_cvtph_epi32 (__m512i src, __mmask16 k, __m256h a);
VCVTPH2DQ __m512i _mm512_maskz_cvtph_epi32 (__mmask16 k, __m256h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

VCVTPH2PD—Convert Packed FP16 Values to FP64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5A /r VCVTPH2PD xmm1{k1}{z}, xmm2/m32/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert packed FP16 values in xmm2/m32/m16bcst to FP64 values, and store result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5A /r VCVTPH2PD ymm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert packed FP16 values in xmm2/m64/m16bcst to FP64 values, and store result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5A /r VCVTPH2PD zmm1{k1}{z}, xmm2/m128/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert packed FP16 values in xmm2/m128/m16bcst to FP64 values, and store result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

This instruction handles both normal and denormal FP16 inputs.

Operation

VCVTPH2PD DEST, SRC

```
VL = 128, 256, or 512
KL := VL/64

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.fp64[j] := Convert_fp16_to_fp64(tsrc)
    ELSE IF *zeroing*:
        DEST.fp64[j] := 0
    // else dest.fp64[j] remains unchanged

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2PD __m512d _mm512_cvt_roundph_pd (__m128h a, int sae);
VCVTPH2PD __m512d _mm512_mask_cvt_roundph_pd (__m512d src, __mmask8 k, __m128h a, int sae);
VCVTPH2PD __m512d _mm512_maskz_cvt_roundph_pd (__mmask8 k, __m128h a, int sae);
VCVTPH2PD __m128d _mm_cvtph_pd (__m128h a);
VCVTPH2PD __m128d _mm_mask_cvtph_pd (__m128d src, __mmask8 k, __m128h a);
VCVTPH2PD __m128d _mm_maskz_cvtph_pd (__mmask8 k, __m128h a);
VCVTPH2PD __m256d _mm256_cvtph_pd (__m128h a);
VCVTPH2PD __m256d _mm256_mask_cvtph_pd (__m256d src, __mmask8 k, __m128h a);
VCVTPH2PD __m256d _mm256_maskz_cvtph_pd (__mmask8 k, __m128h a);
VCVTPH2PD __m512d _mm512_cvtph_pd (__m128h a);
VCVTPH2PD __m512d _mm512_mask_cvtph_pd (__m512d src, __mmask8 k, __m128h a);
VCVTPH2PD __m512d _mm512_maskz_cvtph_pd (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

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

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

Opcode/ 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 FP16 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 FP16 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 AND AVX512F) OR AVX10.1[1]	Convert four packed FP16 values in xmm2/m64 to packed single precision floating-point values in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert eight packed FP16 values in xmm2/m128 to packed single precision floating-point values in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 13 /r VCVTPH2PS zmm1 {k1}{z}, ymm2/m256 {sae}	B	V/V	AVX512F OR AVX10.1[1]	Convert sixteen packed FP16 values in ymm2/m256 to packed single precision floating-point values in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 13 /r VCVTPH2PSX xmm1{k1}{z}, xmm2/m64/m16bcst	C	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert four packed FP16 values in xmm2/m64/m16bcst to four packed single precision floating-point values, and store result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 13 /r VCVTPH2PSX ymm1{k1}{z}, xmm2/m128/m16bcst	C	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight packed single precision floating-point values, and store result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 13 /r VCVTPH2PSX zmm1{k1}{z}, ymm2/m256/m16bcst {sae}	C	V/V	AVX512-FP16 OR AVX10.1[1]	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen packed single precision floating-point values, and store result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

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

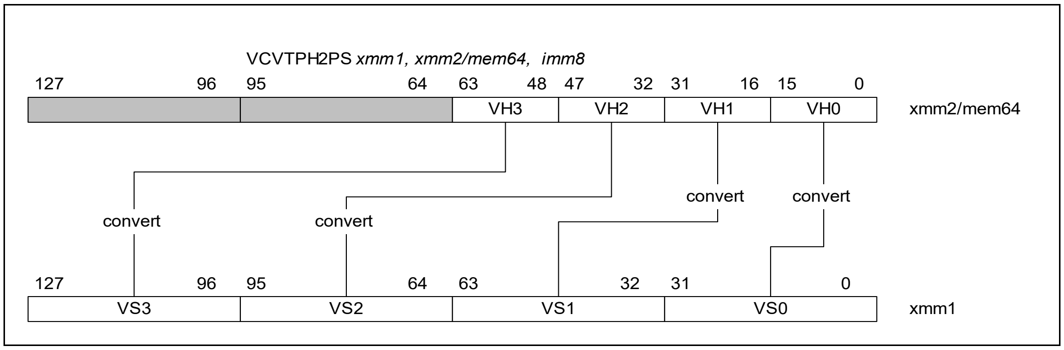


Figure 1-33. VCVTPH2PS (128-bit Version)

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

The instructions associated with AVX512_FP16 always handle FP16 denormal number inputs; denormal inputs are not treated as zero.

Operation

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

VCVTPH2PS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            vCvt_h2s(SRC[k+15:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPH2PS (VEX.256 Encoded Version)

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

VCVTPH2PS (VEX.128 Encoded Version)

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

VCVTPH2PSX DEST, SRC

```
VL = 128, 256, or 512
KL := VL/32

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

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

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

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

VCVTPH2PSX __m512 _mm512_cvtx_roundph_ps (__m256h a, int sae);
VCVTPH2PSX __m512 _mm512_mask_cvtx_roundph_ps (__m512 src, __mmask16 k, __m256h a, int sae);
VCVTPH2PSX __m512 _mm512_maskz_cvtx_roundph_ps (__mmask16 k, __m256h a, int sae);
VCVTPH2PSX __m128 _mm_cvtxph_ps (__m128h a);
VCVTPH2PSX __m128 _mm_mask_cvtxph_ps (__m128 src, __mmask8 k, __m128h a);
VCVTPH2PSX __m128 _mm_maskz_cvtxph_ps (__mmask8 k, __m128h a);
VCVTPH2PSX __m256 _mm256_cvtxph_ps (__m128h a);
VCVTPH2PSX __m256 _mm256_mask_cvtxph_ps (__m256 src, __mmask8 k, __m128h a);
VCVTPH2PSX __m256 _mm256_maskz_cvtxph_ps (__mmask8 k, __m128h a);
VCVTPH2PSX __m512 _mm512_cvtxph_ps (__m256h a);
VCVTPH2PSX __m512 _mm512_mask_cvtxph_ps (__m512 src, __mmask16 k, __m256h a);
VCVTPH2PSX __m512 _mm512_maskz_cvtxph_ps (__mmask16 k, __m256h a);

SIMD Floating-Point Exceptions

VEX-encoded instructions: Invalid.

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

Other Exceptions

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

EVEX-encoded instructions, see Table 2-60, “Type E11 Class Exception Conditions.”

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

Additionally:

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

VCVTPH2QQ—Convert Packed FP16 Values to Signed Quadword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 7B /r VCVTPH2QQ xmm1{k1}{z}, xmm2/m32/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert two packed FP16 values in xmm2/m32/m16bcst to two signed quadword integers, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP5.W0 7B /r VCVTPH2QQ ymm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four signed quadword integers, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP5.W0 7B /r VCVTPH2QQ zmm1{k1}{z}, xmm2/m128/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed quadword integers, and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

The destination elements are updated according to the writemask.

Operation

VCVTPH2QQ DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.qword[j] := Convert_fp16_to_integer64(tsrc)
```

```
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    // else dest.qword[j] remains unchanged

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2QQ __m512i _mm512_cvt_roundph_epi64 (__m128h a, int rounding);
VCVTPH2QQ __m512i _mm512_mask_cvt_roundph_epi64 (__m512i src, __mmask8 k, __m128h a, int rounding);
VCVTPH2QQ __m512i _mm512_maskz_cvt_roundph_epi64 (__mmask8 k, __m128h a, int rounding);
VCVTPH2QQ __m128i _mm_cvtph_epi64 (__m128h a);
VCVTPH2QQ __m128i _mm_mask_cvtph_epi64 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2QQ __m128i _mm_maskz_cvtph_epi64 (__mmask8 k, __m128h a);
VCVTPH2QQ __m256i _mm256_cvtph_epi64 (__m128h a);
VCVTPH2QQ __m256i _mm256_mask_cvtph_epi64 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2QQ __m256i _mm256_maskz_cvtph_epi64 (__mmask8 k, __m128h a);
VCVTPH2QQ __m512i _mm512_cvtph_epi64 (__m128h a);
VCVTPH2QQ __m512i _mm512_mask_cvtph_epi64 (__m512i src, __mmask8 k, __m128h a);
VCVTPH2QQ __m512i _mm512_maskz_cvtph_epi64 (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

VCVTPH2UDQ—Convert Packed FP16 Values to Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 79 /r VCVTPH2UDQ xmm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert four packed FP16 values in xmm2/m64/m16bcst to four unsigned doubleword integers, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 79 /r VCVTPH2UDQ ymm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned doubleword integers, and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 79 /r VCVTPH2UDQ zmm1{k1}{z}, ymm2/m256/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen unsigned doubleword integers, and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

The destination elements are updated according to the writemask.

Operation

VCVTPH2UDQ DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 32

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
            DEST.dword[j] := Convert_fp16_to_unsigned_integer32(tsrc)
```

```
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    // else dest.dword[j] remains unchanged

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2UDQ __m512i _mm512_cvt_roundph_epu32 (__m256h a, int rounding);
VCVTPH2UDQ __m512i _mm512_mask_cvt_roundph_epu32 (__m512i src, __mmask16 k, __m256h a, int rounding);
VCVTPH2UDQ __m512i _mm512_maskz_cvt_roundph_epu32 (__mmask16 k, __m256h a, int rounding);
VCVTPH2UDQ __m128i _mm_cvtph_epu32 (__m128h a);
VCVTPH2UDQ __m128i _mm_mask_cvtph_epu32 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2UDQ __m128i _mm_maskz_cvtph_epu32 (__mmask8 k, __m128h a);
VCVTPH2UDQ __m256i _mm256_cvtph_epu32 (__m128h a);
VCVTPH2UDQ __m256i _mm256_mask_cvtph_epu32 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2UDQ __m256i _mm256_maskz_cvtph_epu32 (__mmask8 k, __m128h a);
VCVTPH2UDQ __m512i _mm512_cvtph_epu32 (__m256h a);
VCVTPH2UDQ __m512i _mm512_mask_cvtph_epu32 (__m512i src, __mmask16 k, __m256h a);
VCVTPH2UDQ __m512i _mm512_maskz_cvtph_epu32 (__mmask16 k, __m256h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

VCVTPH2UQQ—Convert Packed FP16 Values to Unsigned Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 79 /r VCVTPH2UQQ xmm1{k1}{z}, xmm2/m32/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert two packed FP16 values in xmm2/m32/m16bcst to two unsigned quadword integers, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP5.W0 79 /r VCVTPH2UQQ ymm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert four packed FP16 values in xmm2/m64/m16bcst to four unsigned quadword integers, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP5.W0 79 /r VCVTPH2UQQ zmm1{k1}{z}, xmm2/m128/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned quadword integers, and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

The destination elements are updated according to the writemask.

Operation

VCVTPH2UQQ DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.qword[j] := Convert_fp16_to_unsigned_integer64(tsrc)
```

```
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    // else dest.qword[j] remains unchanged

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2UQQ __m512i _mm512_cvt_roundph_epu64 (__m128h a, int rounding);
VCVTPH2UQQ __m512i _mm512_mask_cvt_roundph_epu64 (__m512i src, __mmask8 k, __m128h a, int rounding);
VCVTPH2UQQ __m512i _mm512_maskz_cvt_roundph_epu64 (__mmask8 k, __m128h a, int rounding);
VCVTPH2UQQ __m128i _mm_cvtph_epu64 (__m128h a);
VCVTPH2UQQ __m128i _mm_mask_cvtph_epu64 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2UQQ __m128i _mm_maskz_cvtph_epu64 (__mmask8 k, __m128h a);
VCVTPH2UQQ __m256i _mm256_cvtph_epu64 (__m128h a);
VCVTPH2UQQ __m256i _mm256_mask_cvtph_epu64 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2UQQ __m256i _mm256_maskz_cvtph_epu64 (__mmask8 k, __m128h a);
VCVTPH2UQQ __m512i _mm512_cvtph_epu64 (__m128h a);
VCVTPH2UQQ __m512i _mm512_mask_cvtph_epu64 (__m512i src, __mmask8 k, __m128h a);
VCVTPH2UQQ __m512i _mm512_maskz_cvtph_epu64 (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

VCVTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 7D /r VCVTPH2UW xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert packed FP16 values in xmm2/m128/m16bcst to unsigned word integers, and store the result in xmm1.
EVEX.256.NP.MAP5.W0 7D /r VCVTPH2UW ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert packed FP16 values in ymm2/m256/m16bcst to unsigned word integers, and store the result in ymm1.
EVEX.512.NP.MAP5.W0 7D /r VCVTPH2UW zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert packed FP16 values in zmm2/m512/m16bcst to unsigned word integers, and store the result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

This instruction converts packed FP16 values in the source operand to unsigned word integers in the destination operand.

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

The destination elements are updated according to the writemask.

Operation

VCVTPH2UW DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.word[j] := Convert_fp16_to_unsigned_integer16(tsrc)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    // else dest.word[j] remains unchanged
```

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2UW __m512i _mm512_cvt_roundph_epu16 (__m512h a, int sae);
VCVTPH2UW __m512i _mm512_mask_cvt_roundph_epu16 (__m512i src, __mmask32 k, __m512h a, int sae);
VCVTPH2UW __m512i _mm512_maskz_cvt_roundph_epu16 (__mmask32 k, __m512h a, int sae);
VCVTPH2UW __m128i _mm_cvtph_epu16 (__m128h a);
VCVTPH2UW __m128i _mm_mask_cvtph_epu16 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2UW __m128i _mm_maskz_cvtph_epu16 (__mmask8 k, __m128h a);
VCVTPH2UW __m256i _mm256_cvtph_epu16 (__m256h a);
VCVTPH2UW __m256i _mm256_mask_cvtph_epu16 (__m256i src, __mmask16 k, __m256h a);
VCVTPH2UW __m256i _mm256_maskz_cvtph_epu16 (__mmask16 k, __m256h a);
VCVTPH2UW __m512i _mm512_cvtph_epu16 (__m512h a);
VCVTPH2UW __m512i _mm512_mask_cvtph_epu16 (__m512i src, __mmask32 k, __m512h a);
VCVTPH2UW __m512i _mm512_maskz_cvtph_epu16 (__mmask32 k, __m512h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

VCVTPH2W—Convert Packed FP16 Values to Signed Word Integers

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EVEX.128.66.MAP5.W0 7D /r VCVTPH2W xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert packed FP16 values in xmm2/m128/m16bcst to signed word integers, and store the result in xmm1.
EVEX.256.66.MAP5.W0 7D /r VCVTPH2W ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert packed FP16 values in ymm2/m256/m16bcst to signed word integers, and store the result in ymm1.
EVEX.512.66.MAP5.W0 7D /r VCVTPH2W zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert packed FP16 values in zmm2/m512/m16bcst to signed word integers, and store the result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

The destination elements are updated according to the writemask.

Operation

VCVTPH2W DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.word[j] := Convert_fp16_to_integer16(tsrc)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    // else dest.word[j] remains unchanged
```

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2W __m512i _mm512_cvt_roundph_epi16 (__m512h a, int rounding);
VCVTPH2W __m512i _mm512_mask_cvt_roundph_epi16 (__m512i src, __mmask32 k, __m512h a, int rounding);
VCVTPH2W __m512i _mm512_maskz_cvt_roundph_epi16 (__mmask32 k, __m512h a, int rounding);
VCVTPH2W __m128i _mm_cvtph_epi16 (__m128h a);
VCVTPH2W __m128i _mm_mask_cvtph_epi16 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2W __m128i _mm_maskz_cvtph_epi16 (__mmask8 k, __m128h a);
VCVTPH2W __m256i _mm256_cvtph_epi16 (__m256h a);
VCVTPH2W __m256i _mm256_mask_cvtph_epi16 (__m256i src, __mmask16 k, __m256h a);
VCVTPH2W __m256i _mm256_maskz_cvtph_epi16 (__mmask16 k, __m256h a);
VCVTPH2W __m512i _mm512_cvtph_epi16 (__m512h a);
VCVTPH2W __m512i _mm512_mask_cvtph_epi16 (__m512i src, __mmask32 k, __m512h a);
VCVTPH2W __m512i _mm512_maskz_cvtph_epi16 (__mmask32 k, __m512h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

VCVTPS2PH—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 VCVTPS2PH 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 VCVTPS2PH 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 VCVTPS2PH xmm1/m64 {k1}{z}, xmm2, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VCVTPS2PH xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VCVTPS2PH ymm1/m256 {k1}{z}, zmm2{sae}, imm8	B	V/V	AVX512F OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

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.

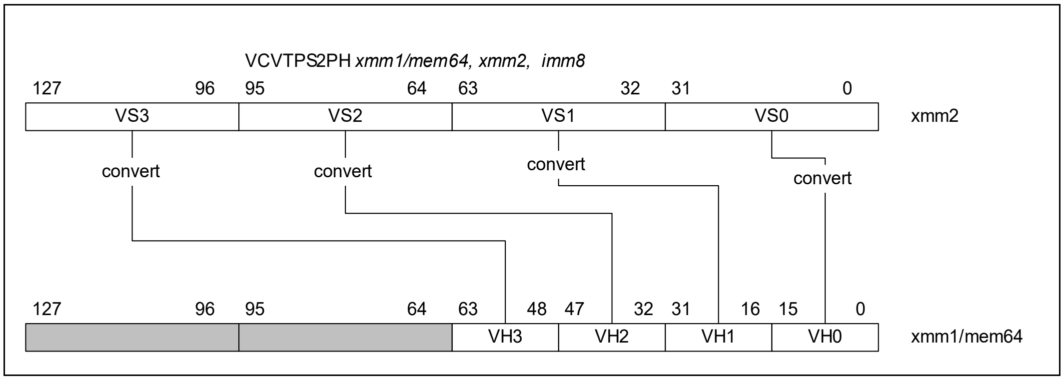


Figure 1-34. VCVTPS2PH (128-bit Version)

The immediate byte defines several bit fields that control rounding operation. The effect and encoding of the RC field are listed in Table 1-10.

Table 1-10. 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 1-10
    RETURN Cvt_Single_Precision_To_Half_Precision_FP_Imm(SRC1[31:0]);
ELSE    ; using MXCSR.RC for rounding control
    RETURN Cvt_Single_Precision_To_Half_Precision_FP_Mxcsr(SRC1[31:0]);
FI;
}
```

VCVTPS2PH (EVEX Encoded Versions) When DEST is a Register

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

VCVTPS2PH (EVEX Encoded Versions) When DEST is Memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 16
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] :=
            vCvt_s2h(SRC[k+31:k])
        ELSE
            *DEST[i+15:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

VCVTPS2PH (VEX.256 Encoded Version)

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

VCVTPS2PH (VEX.128 Encoded Version)

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

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2PH __m256i _mm512_cvtps_ph(__m512 a);
VCVTPS2PH __m256i _mm512_mask_cvtps_ph(__m256i s, __mmask16 k,__m512 a);
VCVTPS2PH __m256i _mm512_maskz_cvtps_ph(__mmask16 k,__m512 a);
VCVTPS2PH __m256i _mm512_cvt_roundps_ph(__m512 a, const int imm);
VCVTPS2PH __m256i _mm512_mask_cvt_roundps_ph(__m256i s, __mmask16 k,__m512 a, const int imm);
VCVTPS2PH __m256i _mm512_maskz_cvt_roundps_ph(__mmask16 k,__m512 a, const int imm);
VCVTPS2PH __m128i _mm256_mask_cvtps_ph(__m128i s, __mmask8 k,__m256 a);
VCVTPS2PH __m128i _mm256_maskz_cvtps_ph(__mmask8 k,__m256 a);
VCVTPS2PH __m128i _mm_mask_cvtps_ph(__m128i s, __mmask8 k,__m128 a);
VCVTPS2PH __m128i _mm_maskz_cvtps_ph(__mmask8 k,__m128 a);
VCVTPS2PH __m128i _mm_cvtps_ph (__m128 m1, const int imm);
VCVTPS2PH __m128i _mm256_cvtps_ph(__m256 m1, const int imm);

SIMD Floating-Point Exceptions

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

Other Exceptions

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

EVEX-encoded instructions, see Table 2-60, “Type E11 Class Exception Conditions.”

Additionally:

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

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 1D /r VCVTPS2PHX xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert four packed single precision floating-point values in xmm2/m128/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP5.W0 1D /r VCVTPS2PHX xmm1{k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert eight packed single precision floating-point values in ymm2/m256/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.66.MAP5.W0 1D /r VCVTPS2PHX ymm1{k1}{z}, zmm2/m512/m32bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert sixteen packed single precision floating-point values in zmm2 /m512/m32bcst to packed FP16 values, and store the result in ymm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

The VCVTPS2PHX instruction supports broadcasting.

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

Operation

VCVTPS2PHX DEST, SRC (AVX512_FP16 Load Version With Broadcast Support)

```
VL = 128, 256, or 512
KL := VL / 32

IF *SRC is a register* and (VL == 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp32[0]
        ELSE
            tsrc := SRC.fp32[j]
        DEST.fp16[j] := Convert_fp32_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
```

```
    // else dest.fp16[j] remains unchanged
DEST[MAXVL-1:VL/2] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2PHX __m256h _mm512_cvtx_roundps_ph (__m512 a, int rounding);
VCVTPS2PHX __m256h _mm512_mask_cvtx_roundps_ph (__m256h src, __mmask16 k, __m512 a, int rounding);
VCVTPS2PHX __m256h _mm512_maskz_cvtx_roundps_ph (__mmask16 k, __m512 a, int rounding);
VCVTPS2PHX __m128h _mm_cvtxps_ph (__m128 a);
VCVTPS2PHX __m128h _mm_mask_cvtxps_ph (__m128h src, __mmask8 k, __m128 a);
VCVTPS2PHX __m128h _mm_maskz_cvtxps_ph (__mmask8 k, __m128 a);
VCVTPS2PHX __m128h _mm256_cvtxps_ph (__m256 a);
VCVTPS2PHX __m128h _mm256_mask_cvtxps_ph (__m128h src, __mmask8 k, __m256 a);
VCVTPS2PHX __m128h _mm256_maskz_cvtxps_ph (__mmask8 k, __m256 a);
VCVTPS2PHX __m256h _mm512_cvtxps_ph (__m512 a);
VCVTPS2PHX __m256h _mm512_mask_cvtxps_ph (__m256h src, __mmask16 k, __m512 a);
VCVTPS2PHX __m256h _mm512_maskz_cvtxps_ph (__mmask16 k, __m512 a);

SIMD Floating-Point Exceptions

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

Other Exceptions

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

Additionally:

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

VCVTPS2QQ—Convert Packed Single Precision Floating-Point Values to Packed Signed 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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

The source operand is a YMM/XMM/XMM (low 64- 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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Single_Precision_To_QuadInteger(SRC[k+31:k])
        ELSE
```

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

VCVTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

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

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2QQ __m512i _mm512_cvtps_epi64(__m512 a);
VCVTPS2QQ __m512i _mm512_mask_cvtps_epi64(__m512i s, __mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_maskz_cvtps_epi64(__mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_cvt_roundps_epi64(__m512 a, int r);
VCVTPS2QQ __m512i _mm512_mask_cvt_roundps_epi64(__m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m512i _mm512_maskz_cvt_roundps_epi64(__mmask16 k, __m512 a, int r);
VCVTPS2QQ __m256i _mm256_cvtps_epi64(__m256 a);
VCVTPS2QQ __m256i _mm256_mask_cvtps_epi64(__m256i s, __mmask8 k, __m256 a);
VCVTPS2QQ __m256i _mm256_maskz_cvtps_epi64(__mmask8 k, __m256 a);
VCVTPS2QQ __m128i _mm_cvtps_epi64(__m128 a);
VCVTPS2QQ __m128i _mm_mask_cvtps_epi64(__m128i s, __mmask8 k, __m128 a);
VCVTPS2QQ __m128i _mm_maskz_cvtps_epi64(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

Additionally:

#UD	If EVEX.vvvv != 1111B.

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

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

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

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

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

VCVTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

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

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

Additionally:

#UD	If EVEX.vvvv != 1111B.

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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

VCVTPS2UQQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Single_Precision_To_UQuadInteger(SRC[k+31:k])
        ELSE
```

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

VCVTPS2UQQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

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

```
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_UQuadInteger(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 Table 2-47, “Type E3 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

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

Opcode/ 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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	Convert eight packed quadword integers from zmm2/m512/m64bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

Operation

VCVTQQ2PD (EVEX2 Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

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

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

VCVTQQ2PD (EVEX Encoded Versions) when SRC Operand is a Memory Source

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

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

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTQQ2PH—Convert Packed Signed Quadword Integers to Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W1 5B /r VCVTQQ2PH xmm1{k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert two packed signed quadword integers in xmm2/m128/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W1 5B /r VCVTQQ2PH xmm1{k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert four packed signed quadword integers in ymm2/m256/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.NP.MAP5.W1 5B /r VCVTQQ2PH xmm1{k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert eight packed signed quadword integers in zmm2/m512/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

Operation

VCVTQQ2PH DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.qword[0]
        ELSE
            tsrc := SRC.qword[j]
        DEST.fp16[j] := Convert_integer64_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
```

```
    // else dest.fp16[j] remains unchanged

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTQQ2PH __m128h _mm512_cvt_roundepi64_ph (__m512i a, int rounding);
VCVTQQ2PH __m128h _mm512_mask_cvt_roundepi64_ph (__m128h src, __mmask8 k, __m512i a, int rounding);
VCVTQQ2PH __m128h _mm512_maskz_cvt_roundepi64_ph (__mmask8 k, __m512i a, int rounding);
VCVTQQ2PH __m128h _mm_cvtepi64_ph (__m128i a);
VCVTQQ2PH __m128h _mm_mask_cvtepi64_ph (__m128h src, __mmask8 k, __m128i a);
VCVTQQ2PH __m128h _mm_maskz_cvtepi64_ph (__mmask8 k, __m128i a);
VCVTQQ2PH __m128h _mm256_cvtepi64_ph (__m256i a);
VCVTQQ2PH __m128h _mm256_mask_cvtepi64_ph (__m128h src, __mmask8 k, __m256i a);
VCVTQQ2PH __m128h _mm256_maskz_cvtepi64_ph (__mmask8 k, __m256i a);
VCVTQQ2PH __m128h _mm512_cvtepi64_ph (__m512i a);
VCVTQQ2PH __m128h _mm512_mask_cvtepi64_ph (__m128h src, __mmask8 k, __m512i a);
VCVTQQ2PH __m128h _mm512_maskz_cvtepi64_ph (__mmask8 k, __m512i a);
```

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

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

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

Opcode/ 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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	Convert eight packed quadword integers from zmm2/mem to eight packed single precision floating-point values in ymm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

Operation

VCVTQQ2PS (EVEX Encoded Versions) When SRC Operand is a Register

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

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

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

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

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTQQ2PS __m256 _mm512_cvtepi64_ps(__m512i a);
VCVTQQ2PS __m256 _mm512_mask_cvtepi64_ps(__m256 s, __mmask16 k, __m512i a);
VCVTQQ2PS __m256 _mm512_maskz_cvtepi64_ps(__mmask16 k, __m512i a);
VCVTQQ2PS __m256 _mm512_cvt_roundepi64_ps(__m512i a, int r);
VCVTQQ2PS __m256 _mm512_mask_cvt_roundepi_ps(__m256 s, __mmask8 k, __m512i a, int r);
VCVTQQ2PS __m256 _mm512_maskz_cvt_roundepi64_ps(__mmask8 k, __m512i a, int r);
VCVTQQ2PS __m128 _mm256_cvtepi64_ps(__m256i a);
VCVTQQ2PS __m128 _mm256_mask_cvtepi64_ps(__m128 s, __mmask8 k, __m256i a);
VCVTQQ2PS __m128 _mm256_maskz_cvtepi64_ps(__mmask8 k, __m256i a);
VCVTQQ2PS __m128 _mm_cvtepi64_ps(__m128i a);
VCVTQQ2PS __m128 _mm_mask_cvtepi64_ps(__m128 s, __mmask8 k, __m128i a);
VCVTQQ2PS __m128 _mm_maskz_cvtepi64_ps(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

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

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTSD2SH—Convert Low FP64 Value to an FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.MAP5.W1 5A /r VCVTSD2SH xmm1{k1}{z}, xmm2, xmm3/m64 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert the low FP64 value in xmm3/m64 to an FP16 value and store the result in the low element of xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

Operation

VCVTSD2SH dest, src1, src2

```
IF *SRC2 is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := Convert_fp64_to_fp16(SRC2.fp64[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSD2SH __m128h _mm_cvt_roundsd_sh (__m128h a, __m128d b, const int rounding);
VCVTSD2SH __m128h _mm_mask_cvt_roundsd_sh (__m128h src, __mmask8 k, __m128h a, __m128d b, const int rounding);
VCVTSD2SH __m128h _mm_maskz_cvt_roundsd_sh (__mmask8 k, __m128h a, __m128d b, const int rounding);
VCVTSD2SH __m128h _mm_cvtsd_sh (__m128h a, __m128d b);
VCVTSD2SH __m128h _mm_mask_cvtsd_sh (__m128h src, __mmask8 k, __m128h a, __m128d b);
VCVTSD2SH __m128h _mm_maskz_cvtsd_sh (__mmask8 k, __m128h a, __m128d b);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

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

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.0F.W0 79 /r VCVTSD2USI r32, xmm1/m64{er}	A	V/V	AVX512F OR AVX10.1[1]	Convert one double precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32.
EVEX.LLIG.F2.0F.W1 79 /r VCVTSD2USI r64, xmm1/m64{er}	A	V/N.E.[2]	AVX512F OR AVX10.1[1]	Convert one double precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.
2. EVEX.W1 in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

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

Description

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

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

Operation

VCVTSD2USI (EVEX Encoded Version)

```
IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode and OperandSize = 64
    THEN    DEST[63:0] := Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0]);
    ELSE    DEST[31:0] := Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0]);
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSD2USI unsigned int _mm_cvtsd_u32(__m128d);
VCVTSD2USI unsigned int _mm_cvt_roundsd_u32(__m128d, int r);
VCVTSD2USI unsigned __int64 _mm_cvtsd_u64(__m128d);
VCVTSD2USI unsigned __int64 _mm_cvt_roundsd_u64(__m128d, int r);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

VCVTSH2SD—Convert Low FP16 Value to an FP64 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 5A /r VCVTSH2SD xmm1{k1}{z}, xmm2, xmm3/m16 {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert the low FP16 value in xmm3/m16 to an FP64 value and store the result in the low element of xmm1 subject to writemask k1. Bits 127:64 of xmm2 are copied to xmm1[127:64].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

Operation

VCVTSH2SD dest, src1, src2

```
IF k1[0] OR *no writemask*:
    DEST.fp64[0] := Convert_fp16_to_fp64(SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp64[0] := 0
// else dest.fp64[0] remains unchanged

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSH2SD __m128d _mm_cvt_roundsh_sd (__m128d a, __m128h b, const int sae);
VCVTSH2SD __m128d _mm_mask_cvt_roundsh_sd (__m128d src, __mmask8 k, __m128d a, __m128h b, const int sae);
VCVTSH2SD __m128d _mm_maskz_cvt_roundsh_sd (__mmask8 k, __m128d a, __m128h b, const int sae);
VCVTSH2SD __m128d _mm_cvtsh_sd (__m128d a, __m128h b);
VCVTSH2SD __m128d _mm_mask_cvtsh_sd (__m128d src, __mmask8 k, __m128d a, __m128h b);
VCVTSH2SD __m128d _mm_maskz_cvtsh_sd (__mmask8 k, __m128d a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

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

VCVTSH2SI—Convert Low FP16 Value to Signed Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 2D /r VCVTSH2SI r32, xmm1/m16 {er}	A	V/V[1]	AVX512-FP16 OR AVX10.1[2]	Convert the low FP16 element in xmm1/m16 to a signed integer and store the result in r32.
EVEX.LLIG.F3.MAP5.W1 2D /r VCVTSH2SI r64, xmm1/m16 {er}	A	V/N.E.	AVX512-FP16 OR AVX10.1[2]	Convert the low FP16 element in xmm1/m16 to a signed integer and store the result in r64.

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

This instruction converts the low FP16 element in the source operand to a signed integer in the destination general purpose register.

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

Operation

VCVTSH2SI dest, src

```
IF *SRC is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF 64-mode and OperandSize == 64:
    DEST.qword := Convert_fp16_to_integer64(SRC.fp16[0])
ELSE:
    DEST.dword := Convert_fp16_to_integer32(SRC.fp16[0])
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSH2SI int _mm_cvt_roundsh_i32 (__m128h a, int rounding);
VCVTSH2SI __int64 _mm_cvt_roundsh_i64 (__m128h a, int rounding);
VCVTSH2SI int _mm_cvtsh_i32 (__m128h a);
VCVTSH2SI __int64 _mm_cvtsh_i64 (__m128h a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

VCVTSH2SS—Convert Low FP16 Value to FP32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.MAP6.W0 13 /r VCVTSH2SS xmm1{k1}{z}, xmm2, xmm3/m16 {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert the low FP16 element in xmm3/m16 to an FP32 value and store in the low element of xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

Operation

```
VCVTSH2SS dest, src1, src2
IF k1[0] OR *no writemask*:
    DEST.fp32[0] := Convert_fp16_to_fp32(SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp32[0] := 0
// else dest.fp32[0] remains unchanged

DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSH2SS __m128 _mm_cvt_roundsh_ss (__m128 a, __m128h b, const int sae);
VCVTSH2SS __m128 _mm_mask_cvt_roundsh_ss (__m128 src, __mmask8 k, __m128 a, __m128h b, const int sae);
VCVTSH2SS __m128 _mm_maskz_cvt_roundsh_ss (__mmask8 k, __m128 a, __m128h b, const int sae);
VCVTSH2SS __m128 _mm_cvtsh_ss (__m128 a, __m128h b);
VCVTSH2SS __m128 _mm_mask_cvtsh_ss (__m128 src, __mmask8 k, __m128 a, __m128h b);
VCVTSH2SS __m128 _mm_maskz_cvtsh_ss (__mmask8 k, __m128 a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

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

VCVTSH2USI—Convert Low FP16 Value to Unsigned Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 79 /r VCVTSH2USI r32, xmm1/m16 {er}	A	V/V[1]	AVX512-FP16 OR AVX10.1[2]	Convert the low FP16 element in xmm1/m16 to an unsigned integer and store the result in r32.
EVEX.LLIG.F3.MAP5.W1 79 /r VCVTSH2USI r64, xmm1/m16 {er}	A	V/N.E.	AVX512-FP16 OR AVX10.1[2]	Convert the low FP16 element in xmm1/m16 to an unsigned integer and store the result in r64.

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

This instruction converts the low FP16 element in the source operand to an unsigned integer in the destination general purpose register.

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

Operation

VCVTSH2USI dest, src

```
// SET_RM() sets the rounding mode used for this instruction.
IF *SRC is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF 64-mode and OperandSize == 64:
    DEST.qword := Convert_fp16_to_unsigned_integer64(SRC.fp16[0])
ELSE:
    DEST.dword := Convert_fp16_to_unsigned_integer32(SRC.fp16[0])
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSH2USI unsigned int _mm_cvt_roundsh_u32 (__m128h a, int sae);
VCVTSH2USI unsigned __int64 _mm_cvt_roundsh_u64 (__m128h a, int rounding);
VCVTSH2USI unsigned int _mm_cvtsh_u32 (__m128h a);
VCVTSH2USI unsigned __int64 _mm_cvtsh_u64 (__m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 2A /r VCVTSI2SH xmm1, xmm2, r32/m32 {er}	A	V/V[1]	AVX512-FP16 OR AVX10.1[2]	Convert the signed doubleword integer in r32/m32 to an FP16 value and store the result in xmm1. Bits 127:16 of xmm2 are copied to xmm1[127:16].
EVEX.LLIG.F3.MAP5.W1 2A /r VCVTSI2SH xmm1, xmm2, r64/m64 {er}	A	V/N.E.	AVX512-FP16 OR AVX10.1[2]	Convert the signed quadword integer in r64/m64 to an FP16 value and store the result in xmm1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits 127:16 of the XMM register destination are copied from corresponding bits in the first source operand. Bits MAXVL-1:128 of the destination register are zeroed.

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

Operation

VCVTSI2SH dest, src1, src2

```
IF *SRC2 is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF 64-mode and OperandSize == 64:
    DEST.fp16[0] := Convert_integer64_to_fp16(SRC2.qword)
ELSE:
    DEST.fp16[0] := Convert_integer32_to_fp16(SRC2.dword)

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSI2SH __m128h _mm_cvt_roundi32_sh (__m128h a, int b, int rounding);
VCVTSI2SH __m128h _mm_cvt_roundi64_sh (__m128h a, __int64 b, int rounding);
VCVTSI2SH __m128h _mm_cvti32_sh (__m128h a, int b);
VCVTSI2SH __m128h _mm_cvti64_sh (__m128h a, __int64 b);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

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

VCVTSS2SH—Convert Low FP32 Value to an FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.MAP5.W0 1D /r VCVTSS2SH xmm1{k1}{z}, xmm2, xmm3/m32 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert low FP32 value in xmm3/m32 to an FP16 value and store in the low element of xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

Operation

VCVTSS2SH dest, src1, src2

```
IF *SRC2 is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := Convert_fp32_to_fp16(SRC2.fp32[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSS2SH __m128h _mm_cvt_roundss_sh (__m128h a, __m128 b, const int rounding);
VCVTSS2SH __m128h _mm_mask_cvt_roundss_sh (__m128h src, __mmask8 k, __m128h a, __m128 b, const int rounding);
VCVTSS2SH __m128h _mm_maskz_cvt_roundss_sh (__mmask8 k, __m128h a, __m128 b, const int rounding);
VCVTSS2SH __m128h _mm_cvtss_sh (__m128h a, __m128 b);
VCVTSS2SH __m128h _mm_mask_cvtss_sh (__m128h src, __mmask8 k, __m128h a, __m128 b);
VCVTSS2SH __m128h _mm_maskz_cvtss_sh (__mmask8 k, __m128h a, __m128 b);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

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

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.0F.W0 79 /r VCVTSS2USI r32, xmm1/m32{er}	A	V/V	AVX512F OR AVX10.1[1]	Convert one single precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32.
EVEX.LLIG.F3.0F.W1 79 /r VCVTSS2USI r64, xmm1/m32{er}	A	V/N.E.[2]	AVX512F OR AVX10.1[1]	Convert one single precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.
2. EVEX.W1 in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

VCVTSS2USI (EVEX Encoded Version)

```
IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0]);
ELSE
    DEST[31:0] := Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2USI unsigned _mm_cvtss_u32(__m128 a);
VCVTSS2USI unsigned _mm_cvt_roundss_u32(__m128 a, int r);
VCVTSS2USI unsigned __int64 _mm_cvtss_u64(__m128 a);
VCVTSS2USI unsigned __int64 _mm_cvt_roundss_u64(__m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

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

Opcode/ 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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	Convert eight packed double precision floating-point values from zmm2/m512 to eight packed quadword integers in zmm1 using truncation with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

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

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

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

VCVTTPD2QQ (EVEX Encoded Version) When SRC Operand is a Memory Source

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

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

Additionally:

#UD	If EVEX.vvvv != 1111B.

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

Opcode Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 78 /r VCVTTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

VCVTTPD2UDQ (EVEX Encoded Versions) When SRC2 Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
```

```
                ELSE                                    ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

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

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

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

Additionally:

#UD	If EVEX.vvvv != 1111B.

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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	Convert eight packed double precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 using truncation with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2UQQ _mm<size>[_mask[z]]_cvtt[_round]pd_epu64
VCVTTPD2UQQ __m512i _mm512_cvttpd_epu64(__m512d a);
VCVTTPD2UQQ __m512i _mm512_mask_cvttpd_epu64(__m512i s, __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i _mm512_maskz_cvttpd_epu64(__mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i _mm512_cvtt_roundpd_epu64(__m512d a, int sae);
VCVTTPD2UQQ __m512i _mm512_mask_cvtt_roundpd_epu64(__m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m512i _mm512_maskz_cvtt_roundpd_epu64(__mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m256i _mm256_mask_cvttpd_epu64(__m256i s, __mmask8 k, __m256d a);
VCVTTPD2UQQ __m256i _mm256_maskz_cvttpd_epu64(__mmask8 k, __m256d a);
VCVTTPD2UQQ __m128i _mm_mask_cvttpd_epu64(__m128i s, __mmask8 k, __m128d a);
VCVTTPD2UQQ __m128i _mm_maskz_cvttpd_epu64(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

Additionally:

#UD	If EVEX.vvvv != 1111B.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.MAP5.W0 5B /r VCVTTPH2DQ xmm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert four packed FP16 values in xmm2/m64/m16bcst to four signed doubleword integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.F3.MAP5.W0 5B /r VCVTTPH2DQ ymm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed doubleword integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.F3.MAP5.W0 5B /r VCVTTPH2DQ zmm1{k1}{z}, ymm2/m256/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen signed doubleword integers, and store the result in zmm1 using truncation subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

The destination elements are updated according to the writemask.

Operation

VCVTTPH2DQ dest, src

```
VL = 128, 256 or 512
KL := VL / 32

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.fp32[j] := Convert_fp16_to_integer32_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.fp32[j] := 0
    // else dest.fp32[j] remains unchanged

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2DQ __m512i _mm512_cvtt_roundph_epi32 (__m256h a, int sae);
VCVTTPH2DQ __m512i _mm512_mask_cvtt_roundph_epi32 (__m512i src, __mmask16 k, __m256h a, int sae);
VCVTTPH2DQ __m512i _mm512_maskz_cvtt_roundph_epi32 (__mmask16 k, __m256h a, int sae);
VCVTTPH2DQ __m128i _mm_cvttph_epi32 (__m128h a);
VCVTTPH2DQ __m128i _mm_mask_cvttph_epi32 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2DQ __m128i _mm_maskz_cvttph_epi32 (__mmask8 k, __m128h a);
VCVTTPH2DQ __m256i _mm256_cvttph_epi32 (__m128h a);
VCVTTPH2DQ __m256i _mm256_mask_cvttph_epi32 (__m256i src, __mmask8 k, __m128h a);
VCVTTPH2DQ __m256i _mm256_maskz_cvttph_epi32 (__mmask8 k, __m128h a);
VCVTTPH2DQ __m512i _mm512_cvttph_epi32 (__m256h a);
VCVTTPH2DQ __m512i _mm512_mask_cvttph_epi32 (__m512i src, __mmask16 k, __m256h a);
VCVTTPH2DQ __m512i _mm512_maskz_cvttph_epi32 (__mmask16 k, __m256h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 7A /r VCVTTPH2QQ xmm1{k1}{z}, xmm2/m32/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert two packed FP16 values in xmm2/m32/m16bcst to two signed quadword integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.66.MAP5.W0 7A /r VCVTTPH2QQ ymm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert four packed FP16 values in xmm2/m64/m16bcst to four signed quadword integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.66.MAP5.W0 7A /r VCVTTPH2QQ zmm1{k1}{z}, xmm2/m128/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed quadword integers, and store the result in zmm1 using truncation subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2QQ dest, src

```
VL = 128, 256 or 512
KL := VL / 64

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.qword[j] := Convert_fp16_to_integer64_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    // else dest.qword[j] remains unchanged

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2QQ __m512i _mm512_cvtt_roundph_epi64 (__m128h a, int sae);
VCVTTPH2QQ __m512i _mm512_mask_cvtt_roundph_epi64 (__m512i src, __mmask8 k, __m128h a, int sae);
VCVTTPH2QQ __m512i _mm512_maskz_cvtt_roundph_epi64 (__mmask8 k, __m128h a, int sae);
VCVTTPH2QQ __m128i _mm_cvttph_epi64 (__m128h a);
VCVTTPH2QQ __m128i _mm_mask_cvttph_epi64 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2QQ __m128i _mm_maskz_cvttph_epi64 (__mmask8 k, __m128h a);
VCVTTPH2QQ __m256i _mm256_cvttph_epi64 (__m128h a);
VCVTTPH2QQ __m256i _mm256_mask_cvttph_epi64 (__m256i src, __mmask8 k, __m128h a);
VCVTTPH2QQ __m256i _mm256_maskz_cvttph_epi64 (__mmask8 k, __m128h a);
VCVTTPH2QQ __m512i _mm512_cvttph_epi64 (__m128h a);
VCVTTPH2QQ __m512i _mm512_mask_cvttph_epi64 (__m512i src, __mmask8 k, __m128h a);
VCVTTPH2QQ __m512i _mm512_maskz_cvttph_epi64 (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 78 /r VCVTTPH2UDQ xmm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert four packed FP16 values in xmm2/m64/m16bcst to four unsigned doubleword integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.NP.MAP5.W0 78 /r VCVTTPH2UDQ ymm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned doubleword integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.NP.MAP5.W0 78 /r VCVTTPH2UDQ zmm1{k1}{z}, ymm2/m256/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen unsigned doubleword integers, and store the result in zmm1 using truncation subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2UDQ dest, src

```
VL = 128, 256 or 512
KL := VL / 32

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.dword[j] := Convert_fp16_to_unsigned_integer32_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    // else dest.dword[j] remains unchanged

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2UDQ __m512i _mm512_cvtt_roundph_epu32 (__m256h a, int sae);
VCVTTPH2UDQ __m512i _mm512_mask_cvtt_roundph_epu32 (__m512i src, __mmask16 k, __m256h a, int sae);
VCVTTPH2UDQ __m512i _mm512_maskz_cvtt_roundph_epu32 (__mmask16 k, __m256h a, int sae);
VCVTTPH2UDQ __m128i _mm_cvttph_epu32 (__m128h a);
VCVTTPH2UDQ __m128i _mm_mask_cvttph_epu32 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2UDQ __m128i _mm_maskz_cvttph_epu32 (__mmask8 k, __m128h a);
VCVTTPH2UDQ __m256i _mm256_cvttph_epu32 (__m128h a);
VCVTTPH2UDQ __m256i _mm256_mask_cvttph_epu32 (__m256i src, __mmask8 k, __m128h a);
VCVTTPH2UDQ __m256i _mm256_maskz_cvttph_epu32 (__mmask8 k, __m128h a);
VCVTTPH2UDQ __m512i _mm512_cvttph_epu32 (__m256h a);
VCVTTPH2UDQ __m512i _mm512_mask_cvttph_epu32 (__m512i src, __mmask16 k, __m256h a);
VCVTTPH2UDQ __m512i _mm512_maskz_cvttph_epu32 (__mmask16 k, __m256h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 78 /r VCVTTPH2UQQ xmm1{k1}{z}, xmm2/m32/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert two packed FP16 values in xmm2/m32/m16bcst to two unsigned quadword integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.66.MAP5.W0 78 /r VCVTTPH2UQQ ymm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert four packed FP16 values in xmm2/m64/m16bcst to four unsigned quadword integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.66.MAP5.W0 78 /r VCVTTPH2UQQ zmm1{k1}{z}, xmm2/m128/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned quadword integers, and store the result in zmm1 using truncation subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2UQQ dest, src

```
VL = 128, 256 or 512
KL := VL / 64

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.qword[j] := Convert_fp16_to_unsigned_integer64_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    // else dest.qword[j] remains unchanged

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2UQQ __m512i _mm512_cvtt_roundph_epu64 (__m128h a, int sae);
VCVTTPH2UQQ __m512i _mm512_mask_cvtt_roundph_epu64 (__m512i src, __mmask8 k, __m128h a, int sae);
VCVTTPH2UQQ __m512i _mm512_maskz_cvtt_roundph_epu64 (__mmask8 k, __m128h a, int sae);
VCVTTPH2UQQ __m128i _mm_cvttph_epu64 (__m128h a);
VCVTTPH2UQQ __m128i _mm_mask_cvttph_epu64 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2UQQ __m128i _mm_maskz_cvttph_epu64 (__mmask8 k, __m128h a);
VCVTTPH2UQQ __m256i _mm256_cvttph_epu64 (__m128h a);
VCVTTPH2UQQ __m256i _mm256_mask_cvttph_epu64 (__m256i src, __mmask8 k, __m128h a);
VCVTTPH2UQQ __m256i _mm256_maskz_cvttph_epu64 (__mmask8 k, __m128h a);
VCVTTPH2UQQ __m512i _mm512_cvttph_epu64 (__m128h a);
VCVTTPH2UQQ __m512i _mm512_mask_cvttph_epu64 (__m512i src, __mmask8 k, __m128h a);
VCVTTPH2UQQ __m512i _mm512_maskz_cvttph_epu64 (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

VCVTTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 7C /r VCVTTPH2UW xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned word integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.NP.MAP5.W0 7C /r VCVTTPH2UW ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen unsigned word integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.NP.MAP5.W0 7C /r VCVTTPH2UW zmm1{k1}{z}, zmm2/m512/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert thirty-two packed FP16 values in zmm2/m512/m16bcst to thirty-two unsigned word integers, and store the result in zmm1 using truncation subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

This instruction converts packed FP16 values in the source operand to unsigned word integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2UW dest, src

```
VL = 128, 256 or 512
KL := VL / 16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.word[j] := Convert_fp16_to_unsigned_integer16_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    // else dest.word[j] remains unchanged

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2UW __m512i _mm512_cvtt_roundph_epu16 (__m512h a, int sae);
VCVTTPH2UW __m512i _mm512_mask_cvtt_roundph_epu16 (__m512i src, __mmask32 k, __m512h a, int sae);
VCVTTPH2UW __m512i _mm512_maskz_cvtt_roundph_epu16 (__mmask32 k, __m512h a, int sae);
VCVTTPH2UW __m128i _mm_cvttph_epu16 (__m128h a);
VCVTTPH2UW __m128i _mm_mask_cvttph_epu16 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2UW __m128i _mm_maskz_cvttph_epu16 (__mmask8 k, __m128h a);
VCVTTPH2UW __m256i _mm256_cvttph_epu16 (__m256h a);
VCVTTPH2UW __m256i _mm256_mask_cvttph_epu16 (__m256i src, __mmask16 k, __m256h a);
VCVTTPH2UW __m256i _mm256_maskz_cvttph_epu16 (__mmask16 k, __m256h a);
VCVTTPH2UW __m512i _mm512_cvttph_epu16 (__m512h a);
VCVTTPH2UW __m512i _mm512_mask_cvttph_epu16 (__m512i src, __mmask32 k, __m512h a);
VCVTTPH2UW __m512i _mm512_maskz_cvttph_epu16 (__mmask32 k, __m512h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

VCVTTPH2W—Convert Packed FP16 Values to Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 7C /r VCVTTPH2W xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed word integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.66.MAP5.W0 7C /r VCVTTPH2W ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen signed word integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.66.MAP5.W0 7C /r VCVTTPH2W zmm1{k1}{z}, zmm2/m512/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert thirty-two packed FP16 values in zmm2/m512/m16bcst to thirty-two signed word integers, and store the result in zmm1 using truncation subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2W dest, src

```
VL = 128, 256 or 512
KL := VL / 16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.word[j] := Convert_fp16_to_integer16_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    // else dest.word[j] remains unchanged

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2W __m512i _mm512_cvtt_roundph_epi16 (__m512h a, int sae);
VCVTTPH2W __m512i _mm512_mask_cvtt_roundph_epi16 (__m512i src, __mmask32 k, __m512h a, int sae);
VCVTTPH2W __m512i _mm512_maskz_cvtt_roundph_epi16 (__mmask32 k, __m512h a, int sae);
VCVTTPH2W __m128i _mm_cvttph_epi16 (__m128h a);
VCVTTPH2W __m128i _mm_mask_cvttph_epi16 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2W __m128i _mm_maskz_cvttph_epi16 (__mmask8 k, __m128h a);
VCVTTPH2W __m256i _mm256_cvttph_epi16 (__m256h a);
VCVTTPH2W __m256i _mm256_mask_cvttph_epi16 (__m256i src, __mmask16 k, __m256h a);
VCVTTPH2W __m256i _mm256_maskz_cvttph_epi16 (__mmask16 k, __m256h a);
VCVTTPH2W __m512i _mm512_cvttph_epi16 (__m512h a);
VCVTTPH2W __m512i _mm512_mask_cvttph_epi16 (__m512i src, __mmask32 k, __m512h a);
VCVTTPH2W __m512i _mm512_maskz_cvttph_epi16 (__mmask32 k, __m512h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

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

Opcode/ 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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

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

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
   i := j * 64
   k := j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] :=
         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] := 0
         FI
```

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

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

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

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2QQ __m512i _mm512_cvttps_epi64(__m256 a);
VCVTTPS2QQ __m512i _mm512_mask_cvttps_epi64(__m512i s, __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i _mm512_maskz_cvttps_epi64(__mmask16 k, __m256 a);
VCVTTPS2QQ __m512i _mm512_cvtt_roundps_epi64(__m256 a, int sae);
VCVTTPS2QQ __m512i _mm512_mask_cvtt_roundps_epi64(__m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m512i _mm512_maskz_cvtt_roundps_epi64(__mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m256i _mm256_mask_cvttps_epi64(__m256i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m256i _mm256_maskz_cvttps_epi64(__mmask8 k, __m128 a);
VCVTTPS2QQ __m128i _mm_mask_cvttps_epi64(__m128i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m128i _mm_maskz_cvttps_epi64(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

Additionally:

#UD	If EVEX.vvvv != 1111B.

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

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EVEX.128.0F.W0 78 /r VCVTTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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.256.0F.W0 78 /r VCVTTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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.512.0F.W0 78 /r VCVTTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}	A	V/V	AVX512F OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

VCVTTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Register

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

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

VCVTTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

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

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

Additionally:

#UD	If EVEX.vvvv != 1111B.

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

Opcode/ 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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

VCVTTPS2UQQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
```

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

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

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

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2UQQ _mm<size>[_mask[z]]_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 Table 2-47, “Type E3 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.0F.W0 78 /r VCVTTSD2USI r32, xmm1/m64{sae}	A	V/V	AVX512F OR AVX10.1[1]	Convert one double precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32 using truncation.
EVEX.LLIG.F2.0F.W1 78 /r VCVTTSD2USI r64, xmm1/m64{sae}	A	V/N.E.[2]	AVX512F OR AVX10.1[1]	Convert one double precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64 using truncation.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.
2. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

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

Description

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

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

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

Operation

VCVTTSD2USI (EVEX Encoded Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSD2USI unsigned int _mm_cvttsd_u32(__m128d);
VCVTTSD2USI unsigned int _mm_cvtt_roundsd_u32(__m128d, int sae);
VCVTTSD2USI unsigned __int64 _mm_cvttsd_u64(__m128d);
VCVTTSD2USI unsigned __int64 _mm_cvtt_roundsd_u64(__m128d, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 2C /r VCVTTSH2SI r32, xmm1/m16 {sae}	A	V/V[1]	AVX512-FP16 OR AVX10.1[2]	Convert FP16 value in the low element of xmm1/m16 to a signed integer and store the result in r32 using truncation.
EVEX.LLIG.F3.MAP5.W1 2C /r VCVTTSH2SI r64, xmm1/m16 {sae}	A	V/N.E.	AVX512-FP16 OR AVX10.1[2]	Convert FP16 value in the low element of xmm1/m16 to a signed integer and store the result in r64 using truncation.

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

This instruction converts the low FP16 element in the source operand to a signed integer in the destination general purpose register.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value is returned.

Operation

VCVTTSH2SI dest, src

```
IF 64-mode and OperandSize == 64:
    DEST.qword := Convert_fp16_to_integer64_truncate(SRC.fp16[0])
ELSE:
    DEST.dword := Convert_fp16_to_integer32_truncate(SRC.fp16[0])
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSH2SI int _mm_cvtt_roundsh_i32 (__m128h a, int sae);
VCVTTSH2SI __int64 _mm_cvtt_roundsh_i64 (__m128h a, int sae);
VCVTTSH2SI int _mm_cvttsh_i32 (__m128h a);
VCVTTSH2SI __int64 _mm_cvttsh_i64 (__m128h a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 78 /r VCVTTSH2USI r32, xmm1/m16 {sae}	A	V/V[1]	AVX512-FP16 OR AVX10.1[2]	Convert FP16 value in the low element of xmm1/m16 to an unsigned integer and store the result in r32 using truncation.
EVEX.LLIG.F3.MAP5.W1 78 /r VCVTTSH2USI r64, xmm1/m16 {sae}	A	V/N.E.	AVX512-FP16 OR AVX10.1[2]	Convert FP16 value in the low element of xmm1/m16 to an unsigned integer and store the result in r64 using truncation.

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.
2. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

This instruction converts the low FP16 element in the source operand to an unsigned integer in the destination general purpose register.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value is returned.

Operation

VCVTTSH2USI dest, src

```
IF 64-mode and OperandSize == 64:
    DEST.qword := Convert_fp16_to_unsigned_integer64_truncate(SRC.fp16[0])
ELSE:
    DEST.dword := Convert_fp16_to_unsigned_integer32_truncate(SRC.fp16[0])
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSH2USI unsigned int _mm_cvtt_roundsh_u32 (__m128h a, int sae);
VCVTTSH2USI unsigned __int64 _mm_cvtt_roundsh_u64 (__m128h a, int sae);
VCVTTSH2USI unsigned int _mm_cvttsh_u32 (__m128h a);
VCVTTSH2USI unsigned __int64 _mm_cvttsh_u64 (__m128h a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.0F.W0 78 /r VCVTTSS2USI r32, xmm1/m32{sae}	A	V/V	AVX512F OR AVX10.1[1]	Convert one single precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32 using truncation.
EVEX.LLIG.F3.0F.W1 78 /r VCVTTSS2USI r64, xmm1/m32{sae}	A	V/N.E.[2]	AVX512F OR AVX10.1[1]	Convert one single precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64 using truncation.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.
2. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

VCVTTSS2USI (EVEX Encoded Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSS2USI unsigned int _mm_cvttss_u32( __m128 a);
VCVTTSS2USI unsigned int _mm_cvtt_roundss_u32( __m128 a, int sae);
VCVTTSS2USI unsigned __int64 _mm_cvttss_u64( __m128 a);
VCVTTSS2USI unsigned __int64 _mm_cvtt_roundss_u64( __m128 a, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

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

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

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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

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

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_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 Table 2-51, “Type E5 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTUDQ2PH—Convert Packed Unsigned Doubleword Integers to Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP5.W0 7A /r VCVTUDQ2PH xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP5.W0 7A /r VCVTUDQ2PH xmm1{k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.F2.MAP5.W0 7A /r VCVTUDQ2PH ymm1{k1}{z}, zmm2/m512/m32bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to packed FP16 values, and store the result in ymm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

Operation

VCVTUDQ2PH dest, src

```
VL = 128, 256 or 512
KL := VL / 32

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.dword[0]
        ELSE
            tsrc := SRC.dword[j]
        DEST.fp16[j] := Convert_unsigned_integer32_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
```

```
    // else dest.fp16[j] remains unchanged

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUDQ2PH __m256h _mm512_cvt_roundepu32_ph (__m512i a, int rounding);
VCVTUDQ2PH __m256h _mm512_mask_cvt_roundepu32_ph (__m256h src, __mmask16 k, __m512i a, int rounding);
VCVTUDQ2PH __m256h _mm512_maskz_cvt_roundepu32_ph (__mmask16 k, __m512i a, int rounding);
VCVTUDQ2PH __m128h _mm_cvtepu32_ph (__m128i a);
VCVTUDQ2PH __m128h _mm_mask_cvtepu32_ph (__m128h src, __mmask8 k, __m128i a);
VCVTUDQ2PH __m128h _mm_maskz_cvtepu32_ph (__mmask8 k, __m128i a);
VCVTUDQ2PH __m128h _mm256_cvtepu32_ph (__m256i a);
VCVTUDQ2PH __m128h _mm256_mask_cvtepu32_ph (__m128h src, __mmask8 k, __m256i a);
VCVTUDQ2PH __m128h _mm256_maskz_cvtepu32_ph (__mmask8 k, __m256i a);
VCVTUDQ2PH __m256h _mm512_cvtepu32_ph (__m512i a);
VCVTUDQ2PH __m256h _mm512_mask_cvtepu32_ph (__m256h src, __mmask16 k, __m512i a);
VCVTUDQ2PH __m256h _mm512_maskz_cvtepu32_ph (__mmask16 k, __m512i a);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

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

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

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to sixteen packed single precision floating-point values in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

Operation

VCVTUDQ2PS (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

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

```
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTUDQ2PS (EVEX Encoded Version) When SRC Operand is a Memory Source

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

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTUDQ2PS __m512 _mm512_cvtepu32_ps( __m512i a);
VCVTUDQ2PS __m512 _mm512_mask_cvtepu32_ps( __m512 s, __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 _mm512_maskz_cvtepu32_ps( __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 _mm512_cvt_roundepu32_ps( __m512i a, int r);
VCVTUDQ2PS __m512 _mm512_mask_cvt_roundepu32_ps( __m512 s, __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m512 _mm512_maskz_cvt_roundepu32_ps( __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m256 _mm256_cvtepu32_ps( __m256i a);
VCVTUDQ2PS __m256 _mm256_mask_cvtepu32_ps( __m256 s, __mmask8 k, __m256i a);
VCVTUDQ2PS __m256 _mm256_maskz_cvtepu32_ps( __mmask8 k, __m256i a);
VCVTUDQ2PS __m128 _mm_cvtepu32_ps( __m128i a);
VCVTUDQ2PS __m128 _mm_mask_cvtepu32_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUDQ2PS __m128 _mm_maskz_cvtepu32_ps( __mmask8 k, __m128i a);
```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

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

Additionally:

#UD	If EVEX.vvvv != 1111B.

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

Opcode/ 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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

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

Description

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

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

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

Operation

VCVTUQQ2PD (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
```

```
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTUQQ2PD (EVEX Encoded Version) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUQQ2PD __m512d _mm512_cvtepu64_ps(__m512i a);
VCVTUQQ2PD __m512d _mm512_mask_cvtepu64_ps(__m512d s, __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_maskz_cvtepu64_ps(__mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_cvt_roundepu64_ps(__m512i a, int r);
VCVTUQQ2PD __m512d _mm512_mask_cvt_roundepu64_ps(__m512d s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m512d _mm512_maskz_cvt_roundepu64_ps(__mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m256d _mm256_cvtepu64_ps(__m256i a);
VCVTUQQ2PD __m256d _mm256_mask_cvtepu64_ps(__m256d s, __mmask8 k, __m256i a);
VCVTUQQ2PD __m256d _mm256_maskz_cvtepu64_ps(__mmask8 k, __m256i a);
VCVTUQQ2PD __m128d _mm_cvtepu64_ps(__m128i a);
VCVTUQQ2PD __m128d _mm_mask_cvtepu64_ps(__m128d s, __mmask8 k, __m128i a);
VCVTUQQ2PD __m128d _mm_maskz_cvtepu64_ps(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTUQQ2PH—Convert Packed Unsigned Quadword Integers to Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP5.W1 7A /r VCVTUQQ2PH xmm1{k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert two packed unsigned doubleword integers from xmm2/m128/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP5.W1 7A /r VCVTUQQ2PH xmm1{k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert four packed unsigned doubleword integers from ymm2/m256/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.F2.MAP5.W1 7A /r VCVTUQQ2PH xmm1{k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert eight packed unsigned doubleword integers from zmm2/m512/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed unsigned quadword integers in the source operand to packed FP16 values in the destination operand. The destination elements are updated according to the writemask.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

VCVTUQQ2PH dest, src

```
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.qword[0]
        ELSE
            tsrc := SRC.qword[j]
        DEST.fp16[j] := Convert_unsigned_integer64_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
```

```
    // else dest.fp16[j] remains unchanged
DEST[MAXVL-1:VL/4] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUQQ2PH __m128h _mm512_cvt_roundepu64_ph (__m512i a, int rounding);
VCVTUQQ2PH __m128h _mm512_mask_cvt_roundepu64_ph (__m128h src, __mmask8 k, __m512i a, int rounding);
VCVTUQQ2PH __m128h _mm512_maskz_cvt_roundepu64_ph (__mmask8 k, __m512i a, int rounding);
VCVTUQQ2PH __m128h _mm_cvtepu64_ph (__m128i a);
VCVTUQQ2PH __m128h _mm_mask_cvtepu64_ph (__m128h src, __mmask8 k, __m128i a);
VCVTUQQ2PH __m128h _mm_maskz_cvtepu64_ph (__mmask8 k, __m128i a);
VCVTUQQ2PH __m128h _mm256_cvtepu64_ph (__m256i a);
VCVTUQQ2PH __m128h _mm256_mask_cvtepu64_ph (__m128h src, __mmask8 k, __m256i a);
VCVTUQQ2PH __m128h _mm256_maskz_cvtepu64_ph (__mmask8 k, __m256i a);
VCVTUQQ2PH __m128h _mm512_cvtepu64_ph (__m512i a);
VCVTUQQ2PH __m128h _mm512_mask_cvtepu64_ph (__m128h src, __mmask8 k, __m512i a);
VCVTUQQ2PH __m128h _mm512_maskz_cvtepu64_ph (__mmask8 k, __m512i a);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed single precision floating-point values in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed unsigned quadword integers in the source operand (second operand) to single precision floating-point values in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTUQQ2PS (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[k+63:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
```

```
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTUQQ2PS (EVEX Encoded Version) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUQQ2PS __m256 _mm512_cvtepu64_ps(__m512i a);
VCVTUQQ2PS __m256 _mm512_mask_cvtepu64_ps(__m256 s, __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 _mm512_maskz_cvtepu64_ps(__mmask8 k, __m512i a);
VCVTUQQ2PS __m256 _mm512_cvt_roundepu64_ps(__m512i a, int r);
VCVTUQQ2PS __m256 _mm512_mask_cvt_roundepu64_ps(__m256 s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m256 _mm512_maskz_cvt_roundepu64_ps(__mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m128 _mm256_cvtepu64_ps(__m256i a);
VCVTUQQ2PS __m128 _mm256_mask_cvtepu64_ps(__m128 s, __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 _mm256_maskz_cvtepu64_ps(__mmask8 k, __m256i a);
VCVTUQQ2PS __m128 _mm_cvtepu64_ps(__m128i a);
VCVTUQQ2PS __m128 _mm_mask_cvtepu64_ps(__m128 s, __mmask8 k, __m128i a);
VCVTUQQ2PS __m128 _mm_maskz_cvtepu64_ps(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTUSI2SD—Convert Unsigned Integer to Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.0F.W0 7B /r VCVTUSI2SD xmm1, xmm2, r/m32	A	V/V	AVX512F OR AVX10.1[1]	Convert one unsigned doubleword integer from r/m32 to one double precision floating-point value in xmm1.
EVEX.LLIG.F2.0F.W1 7B /r VCVTUSI2SD xmm1, xmm2, r/m64{er}	A	V/N.E.[2]	AVX512F OR AVX10.1[1]	Convert one unsigned quadword integer from r/m64 to one double precision floating-point value in xmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.
2. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the second source operand to a double precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

Operation

VCVTUSI2SD (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] := Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] := Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SD __m128d _mm_cvtu32_sd(__m128d s, unsigned a);
VCVTUSI2SD __m128d _mm_cvtu64_sd(__m128d s, unsigned __int64 a);
VCVTUSI2SD __m128d _mm_cvt_roundu64_sd(__m128d s, unsigned __int64 a, int r);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Table 2-48, “Type E3NF Class Exception Conditions” if W1; otherwise, see Table 2-59, “Type E10NF Class Exception Conditions.”

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.LLIG.F3.0F.W0 7B /r VCVTUSI2SS xmm1, xmm2, r/m32{er}	A	V/V	AVX512F OR AVX10.1[1]	Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.
EVEX.LLIG.F3.0F.W1 7B /r VCVTUSI2SS xmm1, xmm2, r/m64{er}	A	V/N.E.[2]	AVX512F OR AVX10.1[1]	Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.
2. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts a unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the source operand (second operand) to a single precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.

Operation

VCVTUSI2SS (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] := Convert_UInteger_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] := Convert_UInteger_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SS __m128 _mm_cvtu32_ss(__m128 s, unsigned a);
VCVTUSI2SS __m128 _mm_cvt_roundu32_ss(__m128 s, unsigned a, int r);
VCVTUSI2SS __m128 _mm_cvtu64_ss(__m128 s, unsigned __int64 a);
VCVTUSI2SS __m128 _mm_cvt_roundu64_ss(__m128 s, unsigned __int64 a, int r);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Table 2-48, “Type E3NF Class Exception Conditions.”

VCVTUW2PH—Convert Packed Unsigned Word Integers to FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP5.W0 7D /r VCVTUW2PH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert eight packed unsigned word integers from xmm2/m128/m16bcst to FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP5.W0 7D /r VCVTUW2PH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert sixteen packed unsigned word integers from ymm2/m256/m16bcst to FP16 values, and store the result in ymm1 subject to writemask k1.
EVEX.512.F2.MAP5.W0 7D /r VCVTUW2PH zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert thirty-two packed unsigned word integers from zmm2/m512/m16bcst to FP16 values, and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed unsigned word integers in the source operand to FP16 values in the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

The destination elements are updated according to the writemask.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

VCVTUW2PH dest, src

```
VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.word[0]
        ELSE
            tsrc := SRC.word[j]
        DEST.fp16[j] := Convert_unsignd_integer16_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUW2PH __m512h _mm512_cvt_roundepu16_ph (__m512i a, int rounding);
VCVTUW2PH __m512h _mm512_mask_cvt_roundepu16_ph (__m512h src, __mmask32 k, __m512i a, int rounding);
VCVTUW2PH __m512h _mm512_maskz_cvt_roundepu16_ph (__mmask32 k, __m512i a, int rounding);
VCVTUW2PH __m128h _mm_cvtepu16_ph (__m128i a);
VCVTUW2PH __m128h _mm_mask_cvtepu16_ph (__m128h src, __mmask8 k, __m128i a);
VCVTUW2PH __m128h _mm_maskz_cvtepu16_ph (__mmask8 k, __m128i a);
VCVTUW2PH __m256h _mm256_cvtepu16_ph (__m256i a);
VCVTUW2PH __m256h _mm256_mask_cvtepu16_ph (__m256h src, __mmask16 k, __m256i a);
VCVTUW2PH __m256h _mm256_maskz_cvtepu16_ph (__mmask16 k, __m256i a);
VCVTUW2PH __m512h _mm512_cvtepu16_ph (__m512i a);
VCVTUW2PH __m512h _mm512_mask_cvtepu16_ph (__m512h src, __mmask32 k, __m512i a);
VCVTUW2PH __m512h _mm512_maskz_cvtepu16_ph (__mmask32 k, __m512i a);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VCVTUW2PH—Convert Packed Unsigned Word Integers to FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP5.W0 7D /r VCVTUW2PH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert eight packed unsigned word integers from xmm2/m128/m16bcst to FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP5.W0 7D /r VCVTUW2PH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert sixteen packed unsigned word integers from ymm2/m256/m16bcst to FP16 values, and store the result in ymm1 subject to writemask k1.
EVEX.512.F2.MAP5.W0 7D /r VCVTUW2PH zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert thirty-two packed unsigned word integers from zmm2/m512/m16bcst to FP16 values, and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed unsigned word integers in the source operand to FP16 values in the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

The destination elements are updated according to the writemask.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

VCVTUW2PH dest, src

```
VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.word[0]
        ELSE
            tsrc := SRC.word[j]
        DEST.fp16[j] := Convert_unsignd_integer16_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUW2PH __m512h _mm512_cvt_roundepu16_ph (__m512i a, int rounding);
VCVTUW2PH __m512h _mm512_mask_cvt_roundepu16_ph (__m512h src, __mmask32 k, __m512i a, int rounding);
VCVTUW2PH __m512h _mm512_maskz_cvt_roundepu16_ph (__mmask32 k, __m512i a, int rounding);
VCVTUW2PH __m128h _mm_cvtepu16_ph (__m128i a);
VCVTUW2PH __m128h _mm_mask_cvtepu16_ph (__m128h src, __mmask8 k, __m128i a);
VCVTUW2PH __m128h _mm_maskz_cvtepu16_ph (__mmask8 k, __m128i a);
VCVTUW2PH __m256h _mm256_cvtepu16_ph (__m256i a);
VCVTUW2PH __m256h _mm256_mask_cvtepu16_ph (__m256h src, __mmask16 k, __m256i a);
VCVTUW2PH __m256h _mm256_maskz_cvtepu16_ph (__mmask16 k, __m256i a);
VCVTUW2PH __m512h _mm512_cvtepu16_ph (__m512i a);
VCVTUW2PH __m512h _mm512_mask_cvtepu16_ph (__m512h src, __mmask32 k, __m512i a);
VCVTUW2PH __m512h _mm512_maskz_cvtepu16_ph (__mmask32 k, __m512i a);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VCVTW2PH—Convert Packed Signed Word Integers to FP16 Values

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EVEX.128.F3.MAP5.W0 7D /r VCVTW2PH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert eight packed signed word integers from xmm2/m128/m16bcst to FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.F3.MAP5.W0 7D /r VCVTW2PH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert sixteen packed signed word integers from ymm2/m256/m16bcst to FP16 values, and store the result in ymm1 subject to writemask k1.
EVEX.512.F3.MAP5.W0 7D /r VCVTW2PH zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert thirty-two packed signed word integers from zmm2/m512/m16bcst to FP16 values, and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed signed word integers in the source operand to FP16 values in the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

The destination elements are updated according to the writemask.

Operation

VCVTW2PH dest, src

```
VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.word[0]
        ELSE
            tsrc := SRC.word[j]
        DEST.fp16[j] := Convert_integer16_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTW2PH __m512h _mm512_cvt_roundepi16_ph (__m512i a, int rounding);
VCVTW2PH __m512h _mm512_mask_cvt_roundepi16_ph (__m512h src, __mmask32 k, __m512i a, int rounding);
VCVTW2PH __m512h _mm512_maskz_cvt_roundepi16_ph (__mmask32 k, __m512i a, int rounding);
VCVTW2PH __m128h _mm_cvtepi16_ph (__m128i a);
VCVTW2PH __m128h _mm_mask_cvtepi16_ph (__m128h src, __mmask8 k, __m128i a);
VCVTW2PH __m128h _mm_maskz_cvtepi16_ph (__mmask8 k, __m128i a);
VCVTW2PH __m256h _mm256_cvtepi16_ph (__m256i a);
VCVTW2PH __m256h _mm256_mask_cvtepi16_ph (__m256h src, __mmask16 k, __m256i a);
VCVTW2PH __m256h _mm256_maskz_cvtepi16_ph (__mmask16 k, __m256i a);
VCVTW2PH __m512h _mm512_cvtepi16_ph (__m512i a);
VCVTW2PH __m512h _mm512_mask_cvtepi16_ph (__m512h src, __mmask32 k, __m512i a);
VCVTW2PH __m512h _mm512_maskz_cvtepi16_ph (__mmask32 k, __m512i a);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes

Opcode/ 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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Compute packed SAD (sum of absolute differences) word results of unsigned bytes from two 32-bit dword elements. Packed SAD word results are calculated in multiples of qword superblocks, producing 4 SAD word results in each 64-bit superblock of the destination register.

Within each super block of packed word results, the SAD results from two 32-bit dword elements are calculated as follows:

- The lower two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from an intermediate vector with a stationary dword element in the corresponding qword superblock of the first source operand. The intermediate vector, see “Tmp1” in Figure 1-35, 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 2and 3 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 4.
- The intermediate vector is constructed in 128-bits lanes. Within each 128-bit lane, each dword element of the intermediate vector is selected by a two-bit field within the imm8 byte on the corresponding 128-bits of the second source operand. The imm8 byte serves as dword shuffle control within each 128-bit lanes of the intermediate vector and the second source operand, similarly to PSHUFD.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1 at 16-bit word granularity.

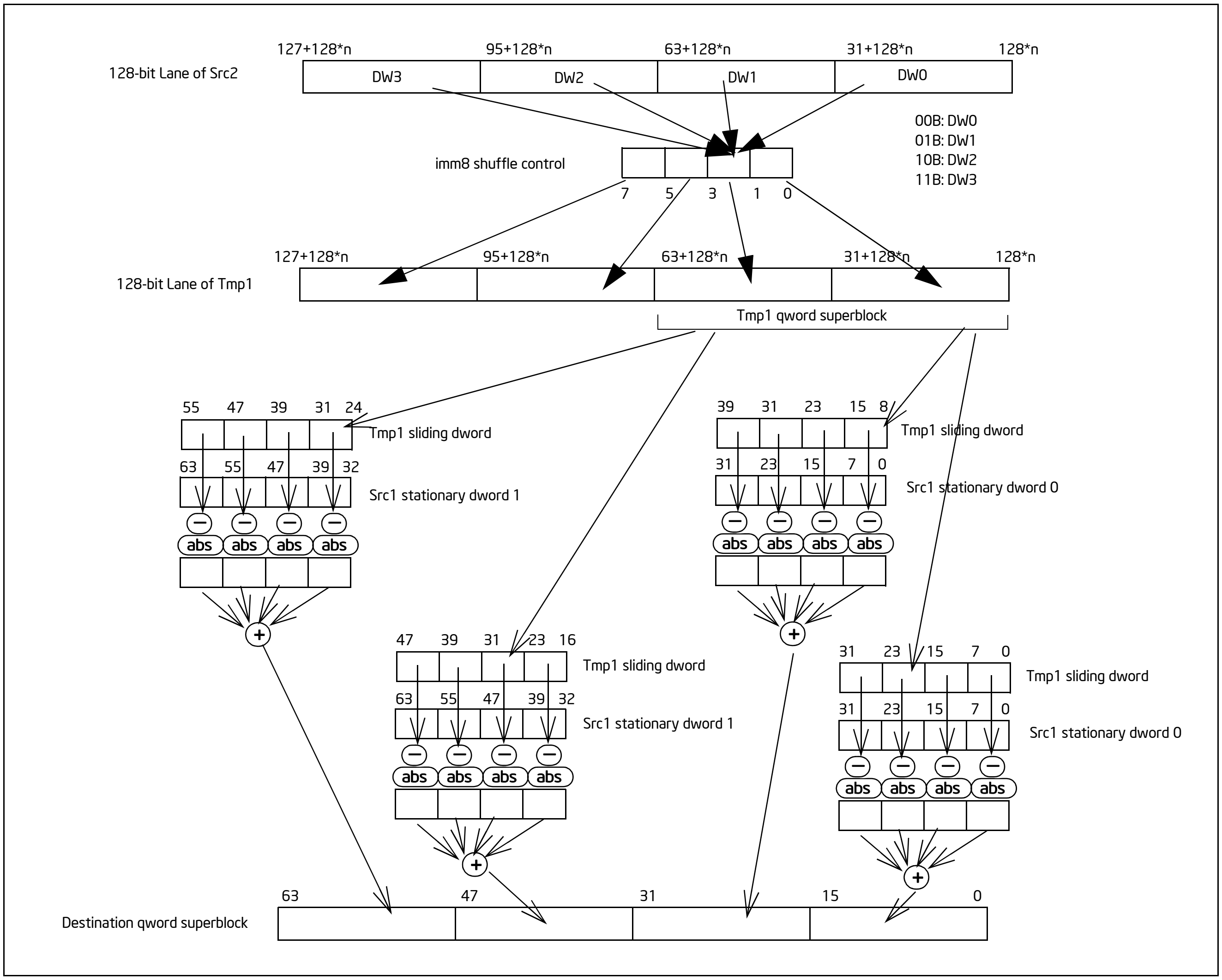


Figure 1-35. 64-bit Super Block of SAD Operation in VDBPSADBW

Operation

VDBPSADBW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
Selection of quadruplets:
FOR I = 0 to VL step 128
    TMP1[I+31:I] := select (SRC2[I+127: I], imm8[1:0])
    TMP1[I+63: I+32] := select (SRC2[I+127: I], imm8[3:2])
    TMP1[I+95: I+64] := select (SRC2[I+127: I], imm8[5:4])
    TMP1[I+127: I+96]  := select (SRC2[I+127: I], imm8[7:6])
END FOR
```

```
SAD of quadruplets:

FOR I =0 to VL step 64
    TMP_DEST[I+15:I] := ABS(SRC1[I+7: I] - TMP1[I+7: I]) +
        ABS(SRC1[I+15: I+8]- TMP1[I+15: I+8]) +
        ABS(SRC1[I+23: I+16]- TMP1[I+23: I+16]) +
        ABS(SRC1[I+31: I+24]- TMP1[I+31: I+24])

    TMP_DEST[I+31: I+16] := ABS(SRC1[I+7: I] - TMP1[I+15: I+8]) +
        ABS(SRC1[I+15: I+8]- TMP1[I+23: I+16]) +
        ABS(SRC1[I+23: I+16]- TMP1[I+31: I+24]) +
        ABS(SRC1[I+31: I+24]- TMP1[I+39: I+32])
    TMP_DEST[I+47: I+32] := ABS(SRC1[I+39: I+32] - TMP1[I+23: I+16]) +
        ABS(SRC1[I+47: I+40]- TMP1[I+31: I+24]) +
        ABS(SRC1[I+55: I+48]- TMP1[I+39: I+32]) +
        ABS(SRC1[I+63: I+56]- TMP1[I+47: I+40])

    TMP_DEST[I+63: I+48] := ABS(SRC1[I+39: I+32] - TMP1[I+31: I+24]) +
        ABS(SRC1[I+47: I+40] - TMP1[I+39: I+32]) +
        ABS(SRC1[I+55: I+48] - TMP1[I+47: I+40]) +
        ABS(SRC1[I+63: I+56] - TMP1[I+55: I+48])
ENDFOR

FOR j :=  0 TO KL-1
    i :=    j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] :=  TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] :=  0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] :=  0
```

Intel C/C++ Compiler Intrinsic Equivalent

VDBPSADBW __m512i _mm512_dbsad_epu8(__m512i a, __m512i b int imm8);
VDBPSADBW __m512i _mm512_mask_dbsad_epu8(__m512i s, __mmask32 m, __m512i a, __m512i b int imm8);
VDBPSADBW __m512i _mm512_maskz_dbsad_epu8(__mmask32 m, __m512i a, __m512i b int imm8);
VDBPSADBW __m256i _mm256_dbsad_epu8(__m256i a, __m256i b int imm8);
VDBPSADBW __m256i _mm256_mask_dbsad_epu8(__m256i s, __mmask16 m, __m256i a, __m256i b int imm8);
VDBPSADBW __m256i _mm256_maskz_dbsad_epu8(__mmask16 m, __m256i a, __m256i b int imm8);
VDBPSADBW __m128i _mm_dbsad_epu8(__m128i a, __m128i b int imm8);
VDBPSADBW __m128i _mm_mask_dbsad_epu8(__m128i s, __mmask8 m, __m128i a, __m128i b int imm8);
VDBPSADBW __m128i _mm_maskz_dbsad_epu8(__mmask8 m, __m128i a, __m128i b int imm8);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

VDIVPH—Divide Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5E /r VDIVPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Divide packed FP16 values in xmm2 by packed FP16 values in xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5E /r VDIVPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Divide packed FP16 values in ymm2 by packed FP16 values in ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5E /r VDIVPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Divide packed FP16 values in zmm2 by packed FP16 values in zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction divides packed FP16 values from the first source operand by the corresponding elements in the second source operand, storing the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

Operation

VDIVPH (EVEX Encoded Versions) When SRC2 Operand is a Register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.fp16[j] := SRC1.fp16[j] / SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VDIVPH (EVEX Encoded Versions) When SRC2 Operand is a Memory Source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            DEST.fp16[j] := SRC1.fp16[j] / SRC2.fp16[0]
        ELSE:
            DEST.fp16[j] := SRC1.fp16[j] / SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VDIVPH __m128h _mm_div_ph (__m128h a, __m128h b);
VDIVPH __m128h _mm_mask_div_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VDIVPH __m128h _mm_maskz_div_ph (__mmask8 k, __m128h a, __m128h b);
VDIVPH __m256h _mm256_div_ph (__m256h a, __m256h b);
VDIVPH __m256h _mm256_mask_div_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VDIVPH __m256h _mm256_maskz_div_ph (__mmask16 k, __m256h a, __m256h b);
VDIVPH __m512h _mm512_div_ph (__m512h a, __m512h b);
VDIVPH __m512h _mm512_mask_div_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VDIVPH __m512h _mm512_maskz_div_ph (__mmask32 k, __m512h a, __m512h b);
VDIVPH __m512h _mm512_div_round_ph (__m512h a, __m512h b, int rounding);
VDIVPH __m512h _mm512_mask_div_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VDIVPH __m512h _mm512_maskz_div_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal, Zero.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VDIVSH—Divide Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 5E /r VDIVSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Divide low FP16 value in xmm2 by low FP16 value in xmm3/m16, and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction divides the low FP16 value from the first source operand by the corresponding value in the second source operand, storing the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VDIVSH (EVEX Encoded Versions)

```
IF EVEX.b = 1 and SRC2 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := SRC1.fp16[0] / SRC2.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VDIVSH __m128h _mm_div_round_sh (__m128h a, __m128h b, int rounding);
VDIVSH __m128h _mm_mask_div_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
VDIVSH __m128h _mm_maskz_div_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
VDIVSH __m128h _mm_div_sh (__m128h a, __m128h b);
VDIVSH __m128h _mm_mask_div_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VDIVSH __m128h _mm_maskz_div_sh (__mmask8 k, __m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal, Zero.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VDPBF16PS—Dot Product of BF16 Pairs Accumulated Into Packed Single Precision

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 52 /r VDPBF16PS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512_BF16 AND AVX512VL) OR AVX10.1[1]	Multiply BF16 pairs from xmm2 and xmm3/m128, and accumulate the resulting packed single precision results in xmm1 with writemask k1.
EVEX.256.F3.0F38.W0 52 /r VDPBF16PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512_BF16 AND AVX512VL) OR AVX10.1[1]	Multiply BF16 pairs from ymm2 and ymm3/m256, and accumulate the resulting packed single precision results in ymm1 with writemask k1.
EVEX.512.F3.0F38.W0 52 /r VDPBF16PS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	(AVX512_BF16 AND AVX512F) OR AVX10.1[1]	Multiply BF16 pairs from zmm2 and zmm3/m512, and accumulate the resulting packed single precision results in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD dot-product of two BF16 pairs and accumulates into a packed single precision register.

“Round to nearest even” rounding mode is used when doing each accumulation of the FMA. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

NaN propagation priorities are described in Table 1-7.

Table 1-7. NaN Propagation Priorities

NaN Priority	Description	Comments
1	src1 low is NaN	Lower part has priority over upper part, i.e., it overrides the upper part.
2	src2 low is NaN	
3	src1 high is NaN	Upper part may be overridden if lower has NaN.
4	src2 high is NaN	
5	srcdest is NaN	Dest is propagated if no NaN is encountered by src2.

Operation

```
Define make_fp32(x):
    // The x parameter is bfloat16. Pack it in to upper 16b of a dword. The bit pattern is a legal fp32 value. Return that bit pattern.
    dword := 0
    dword[31:16] := x
    RETURN dword
```

```
VDPBF16PS srcdest, src1, src2
VL = (128, 256, 512)
KL = VL/32

origdest := srcdest
FOR i := 0 to KL-1:
    IF k1[ i ] or *no writemask*:
        IF src2 is memory and evex.b == 1:
            t := src2.dword[0]
        ELSE:
            t := src2.dword[ i ]

        // FP32 FMA with daz in, ftz out and RNE rounding. MXCSR neither consulted nor updated.

        srcdest.fp32[ i ] += make_fp32(src1.bfloat16[2*i+1]) * make_fp32(t.bfloat[1])
        srcdest.fp32[ i ] += make_fp32(src1.bfloat16[2*i+0]) * make_fp32(t.bfloat[0])

    ELSE IF *zeroing*:
        srcdest.dword[ i ] := 0
    ELSE: // merge masking, dest element unchanged
        srcdest.dword[ i ] := origdest.dword[ i ]

srcdest[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VDPBF16PS __m128 _mm_dpbf16_ps(__m128, __m128bh, __m128bh);
VDPBF16PS __m128 _mm_mask_dpbf16_ps(__m128, __mmask8, __m128bh, __m128bh);
VDPBF16PS __m128 _mm_maskz_dpbf16_ps(__mmask8, __m128, __m128bh, __m128bh);
VDPBF16PS __m256 _mm256_dpbf16_ps(__m256, __m256bh, __m256bh);
VDPBF16PS __m256 _mm256_mask_dpbf16_ps(__m256, __mmask8, __m256bh, __m256bh);
VDPBF16PS __m256 _mm256_maskz_dpbf16_ps(__mmask8, __m256, __m256bh, __m256bh);
VDPBF16PS __m512 _mm512_dpbf16_ps(__m512, __m512bh, __m512bh);
VDPBF16PS __m512 _mm512_mask_dpbf16_ps(__m512, __mmask16, __m512bh, __m512bh);
VDPBF16PS __m512 _mm512_maskz_dpbf16_ps(__mmask16, __m512, __m512bh, __m512bh);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”

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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Expand packed double precision floating-point values from zmm2/m512 to zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Expand (load) up to 8/4/2, contiguous, double precision floating-point values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VEXPANDPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+63:i] := SRC[k+63:k];
            k := k + 64
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
```

```
                THEN DEST[i+63:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VEXPANDPD __m512d _mm512_mask_expand_pd(__m512d s, __mmask8 k, __m512d a);
VEXPANDPD __m512d _mm512_maskz_expand_pd(__mmask8 k, __m512d a);
VEXPANDPD __m512d _mm512_mask_expandloadu_pd(__m512d s, __mmask8 k, void * a);
VEXPANDPD __m512d _mm512_maskz_expandloadu_pd(__mmask8 k, void * a);
VEXPANDPD __m256d _mm256_mask_expand_pd(__m256d s, __mmask8 k, __m256d a);
VEXPANDPD __m256d _mm256_maskz_expand_pd(__mmask8 k, __m256d a);
VEXPANDPD __m256d _mm256_mask_expandloadu_pd(__m256d s, __mmask8 k, void * a);
VEXPANDPD __m256d _mm256_maskz_expandloadu_pd(__mmask8 k, void * a);
VEXPANDPD __m128d _mm_mask_expand_pd(__m128d s, __mmask8 k, __m128d a);
VEXPANDPD __m128d _mm_maskz_expand_pd(__mmask8 k, __m128d a);
VEXPANDPD __m128d _mm_mask_expandloadu_pd(__m128d s, __mmask8 k, void * a);
VEXPANDPD __m128d _mm_maskz_expandloadu_pd(__mmask8 k, void * a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VEXPANDPS—Load Sparse Packed Single Precision Floating-Point Values From Dense Memory

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Expand packed single precision floating-point values from zmm2/m512 to zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Expand (load) up to 16/8/4, contiguous, single precision floating-point values of the input vector in the source operand (the second operand) to sparse elements of the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask k1 selects the destination elements (a partial vector or sparse elements if less than 16 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VEXPANDPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] := SRC[k+31:k];
            k := k + 32
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
```

```
                DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VEXPANDPS __m512 _mm512_mask_expand_ps(__m512 s, __mmask16 k, __m512 a);
VEXPANDPS __m512 _mm512_maskz_expand_ps(__mmask16 k, __m512 a);
VEXPANDPS __m512 _mm512_mask_expandloadu_ps(__m512 s, __mmask16 k, void * a);
VEXPANDPS __m512 _mm512_maskz_expandloadu_ps(__mmask16 k, void * a);
VEXPANDPD __m256 _mm256_mask_expand_ps(__m256 s, __mmask8 k, __m256 a);
VEXPANDPD __m256 _mm256_maskz_expand_ps(__mmask8 k, __m256 a);
VEXPANDPD __m256 _mm256_mask_expandloadu_ps(__m256 s, __mmask8 k, void * a);
VEXPANDPD __m256 _mm256_maskz_expandloadu_ps(__mmask8 k, void * a);
VEXPANDPD __m128 _mm_mask_expand_ps(__m128 s, __mmask8 k, __m128 a);
VEXPANDPD __m128 _mm_maskz_expand_ps(__mmask8 k, __m128 a);
VEXPANDPD __m128 _mm_mask_expandloadu_ps(__m128 s, __mmask8 k, void * a);
VEXPANDPD __m128 _mm_maskz_expandloadu_ps(__mmask8 k, void * a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	Extract 256 bits of packed double precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
B	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
C	Tuple4	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
D	Tuple8	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

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
```

VEXTRACTF64x2 (EVEX Encoded Versions) When Destination is a Register

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;

FOR j := 0 TO 1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                   ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*             ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:128] := 0
```

VEXTRACTF64x2 (EVEX Encoded Versions) When Destination is Memory

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
```

```
            11: TMP_DEST[127:0] := SRC1[511:384]
        ESAC.
FI;

FOR j := 0 TO 1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

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
                    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 Table 2-23, “Type 6 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-54, “Type E6NF Class Exception Conditions.”

Additionally:

#UD	IF VEX.L = 0.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract Packed Integer Values

Opcode/ Instruction	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	Extract 256 bits of quad-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
B	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
C	Tuple4	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
D	Tuple8	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

VEXTRACTI128/VEXTRACTI32x4 and VEXTRACTI64x2 extract 128-bits of doubleword integer values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTI32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTI64x2: The low 128-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEXTRACTI32x8 and VEXTRACTI64x4 extract 256-bits of quadword integer values from the source operand (the second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor The destination may be either a vector register or a 256-bit memory location.

VEXTRACTI32x8: The low 256-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTI64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The high 7 bits (6 bits in EVEX.512) of the immediate are ignored.

If VEXTRACTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VEXTRACTI32x4 (EVEX encoded versions) when destination is a register

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;
FOR j := 0 TO 3
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:128] := 0
```

VEXTRACTI32x4 (EVEX encoded versions) when destination is memory

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
```

```
            01: TMP_DEST[127:0] := SRC1[255:128]
            10: TMP_DEST[127:0] := SRC1[383:256]
            11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;

FOR j := 0 TO 3
    i := j * 32
    IF k1[j] OR *no writemask*
            THEN DEST[i+31:i] := TMP_DEST[i+31:i]
            ELSE *DEST[i+31:i] remains unchanged*           ; merging-masking
    FI;
ENDFOR
```

VEXTRACTI64x2 (EVEX encoded versions) when destination is a register

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
            0: TMP_DEST[127:0] := SRC1[127:0]
            1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
            00: TMP_DEST[127:0] := SRC1[127:0]
            01: TMP_DEST[127:0] := SRC1[255:128]
            10: TMP_DEST[127:0] := SRC1[383:256]
            11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;

FOR j := 0 TO 1
    i := j * 64
    IF k1[j] OR *no writemask*
            THEN DEST[i+63:i] := TMP_DEST[i+63:i]
            ELSE
                    IF *merging-masking*                        ; merging-masking
                            THEN *DEST[i+63:i] remains unchanged*
                            ELSE *zeroing-masking*                  ; zeroing-masking
                                    DEST[i+63:i] := 0
                    FI
    FI;
ENDFOR
DEST[MAXVL-1:128] := 0
```

VEXTRACTI64x2 (EVEX encoded versions) when destination is memory

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
            0: TMP_DEST[127:0] := SRC1[127:0]
            1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
```

```
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;

FOR j := 0 TO 1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*            ; merging-masking
    FI;
ENDFOR
```

VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is a register

```
VL = 512
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 7
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                        ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:256] := 0
```

VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is memory

```
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 7
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*            ; merging-masking
    FI;
ENDFOR
```

VEXTRACTI64x4 (EVEX.512 encoded version) when destination is a register

```
VL = 512
CASE (imm8[0]) OF
```

```
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 3
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:256] := 0
```

VEXTRACTI64x4 (EVEX.512 encoded version) when destination is memory

```
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.
FOR j := 0 TO 3
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

VEXTRACTI128 (memory destination form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] := SRC1[127:0]
    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 Table 2-23, “Type 6 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-54, “Type E6NF Class Exception Conditions.”

Additionally:

#UD	IF VEX.L = 0.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VFCMADDCPH/VFMADDCPH—Complex Multiply and Accumulate FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP6.W0 56 /r VFCMADDCPH xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Complex multiply a pair of FP16 values from xmm2 and xmm3/m128/m32bcst, add to xmm1 and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP6.W0 56 /r VFCMADDCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Complex multiply a pair of FP16 values from ymm2 and ymm3/m256/m32bcst, add to ymm1 and store the result in ymm1 subject to writemask k1.
EVEX.512.F2.MAP6.W0 56 /r VFCMADDCPH zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Complex multiply a pair of FP16 values from zmm2 and zmm3/m512/m32bcst, add to zmm1 and store the result in zmm1 subject to writemask k1.
EVEX.128.F3.MAP6.W0 56 /r VFMADDCPH xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Complex multiply a pair of FP16 values from xmm2 and the complex conjugate of xmm3/m128/m32bcst, add to xmm1 and store the result in xmm1 subject to writemask k1.
EVEX.256.F3.MAP6.W0 56 /r VFMADDCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Complex multiply a pair of FP16 values from ymm2 and the complex conjugate of ymm3/m256/m32bcst, add to ymm1 and store the result in ymm1 subject to writemask k1.
EVEX.512.F3.MAP6.W0 56 /r VFMADDCPH zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Complex multiply a pair of FP16 values from zmm2 and the complex conjugate of zmm3/m512/m32bcst, add to zmm1 and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a complex multiply and accumulate operation. There are normal and complex conjugate forms of the operation.

The broadcasting and masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

Operation

VFMADDCPH dest{k1}, src1, src2 (AVX512)

```
VL = 128, 256, 512
KL := VL / 32

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF broadcasting and src2 is memory:
            tsrc2.fp16[2*i+0] := src2.fp16[0]
            tsrc2.fp16[2*i+1] := src2.fp16[1]
        ELSE:
            tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
            tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        tmp[2*i+0] := dest.fp16[2*i+0] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
        tmp[2*i+1] := dest.fp16[2*i+1] + src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        // non-conjugate version subtracts even term
        dest.fp16[2*i+0] := tmp[2*i+0] - src1.fp16[2*i+1] * tsrc2.fp16[2*i+1]
        dest.fp16[2*i+1] := tmp[2*i+1] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+1]
    ELSE IF *zeroing*:
        dest.fp16[2*i+0] := 0
        dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0
```

VFMADDCPH dest{k1}, src1, src2 (AVX512)

```
VL = 128, 256, 512
KL := VL / 32

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF broadcasting and src2 is memory:
            tsrc2.fp16[2*i+0] := src2.fp16[0]
            tsrc2.fp16[2*i+1] := src2.fp16[1]
        ELSE:
            tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
            tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        tmp[2*i+0] := dest.fp16[2*i+0] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
        tmp[2*i+1] := dest.fp16[2*i+1] + src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        // conjugate version subtracts odd final term
        dest.fp16[2*i+0] := tmp[2*i+0] + src1.fp16[2*i+1] * tsrc2.fp16[2*i+1]
        dest.fp16[2*i+1] := tmp[2*i+1] - src1.fp16[2*i+0] * tsrc2.fp16[2*i+1]
    ELSE IF *zeroing*:
```

```
        dest.fp16[2*i+0] := 0
        dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFCMADDCPH __m128h _mm_fcmadd_pch (__m128h a, __m128h b, __m128h c);
VFCMADDCPH __m128h _mm_mask_fcmadd_pch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFCMADDCPH __m128h _mm_mask3_fcmadd_pch (__m128h a, __m128h b, __m128h c, __mmask8 k);
VFCMADDCPH __m128h _mm_maskz_fcmadd_pch (__mmask8 k, __m128h a, __m128h b, __m128h c);
VFCMADDCPH __m256h _mm256_fcmadd_pch (__m256h a, __m256h b, __m256h c);
VFCMADDCPH __m256h _mm256_mask_fcmadd_pch (__m256h a, __mmask8 k, __m256h b, __m256h c);
VFCMADDCPH __m256h _mm256_mask3_fcmadd_pch (__m256h a, __m256h b, __m256h c, __mmask8 k);
VFCMADDCPH __m256h _mm256_maskz_fcmadd_pch (__mmask8 k, __m256h a, __m256h b, __m256h c);
VFCMADDCPH __m512h _mm512_fcmadd_pch (__m512h a, __m512h b, __m512h c);
VFCMADDCPH __m512h _mm512_mask_fcmadd_pch (__m512h a, __mmask16 k, __m512h b, __m512h c);
VFCMADDCPH __m512h _mm512_mask3_fcmadd_pch (__m512h a, __m512h b, __m512h c, __mmask16 k);
VFCMADDCPH __m512h _mm512_maskz_fcmadd_pch (__mmask16 k, __m512h a, __m512h b, __m512h c);
VFCMADDCPH __m512h _mm512_fcmadd_round_pch (__m512h a, __m512h b, __m512h c, const int rounding);
VFCMADDCPH __m512h _mm512_mask_fcmadd_round_pch (__m512h a, __mmask16 k, __m512h b, __m512h c, const int rounding);
VFCMADDCPH __m512h _mm512_mask3_fcmadd_round_pch (__m512h a, __m512h b, __m512h c, __mmask16 k, const int rounding);
VFCMADDCPH __m512h _mm512_maskz_fcmadd_round_pch (__mmask16 k, __m512h a, __m512h b, __m512h c, const int rounding);

VFMADDCPH __m128h _mm_fmadd_pch (__m128h a, __m128h b, __m128h c);
VFMADDCPH __m128h _mm_mask_fmadd_pch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFMADDCPH __m128h _mm_mask3_fmadd_pch (__m128h a, __m128h b, __m128h c, __mmask8 k);
VFMADDCPH __m128h _mm_maskz_fmadd_pch (__mmask8 k, __m128h a, __m128h b, __m128h c);
VFMADDCPH __m256h _mm256_fmadd_pch (__m256h a, __m256h b, __m256h c);
VFMADDCPH __m256h _mm256_mask_fmadd_pch (__m256h a, __mmask8 k, __m256h b, __m256h c);
VFMADDCPH __m256h _mm256_mask3_fmadd_pch (__m256h a, __m256h b, __m256h c, __mmask8 k);
VFMADDCPH __m256h _mm256_maskz_fmadd_pch (__mmask8 k, __m256h a, __m256h b, __m256h c);
VFMADDCPH __m512h _mm512_fmadd_pch (__m512h a, __m512h b, __m512h c);
VFMADDCPH __m512h _mm512_mask_fmadd_pch (__m512h a, __mmask16 k, __m512h b, __m512h c);
VFMADDCPH __m512h _mm512_mask3_fmadd_pch (__m512h a, __m512h b, __m512h c, __mmask16 k);
VFMADDCPH __m512h _mm512_maskz_fmadd_pch (__mmask16 k, __m512h a, __m512h b, __m512h c);
VFMADDCPH __m512h _mm512_fmadd_round_pch (__m512h a, __m512h b, __m512h c, const int rounding);
VFMADDCPH __m512h _mm512_mask_fmadd_round_pch (__m512h a, __mmask16 k, __m512h b, __m512h c, const int rounding);
VFMADDCPH __m512h _mm512_mask3_fmadd_round_pch (__m512h a, __m512h b, __m512h c, __mmask16 k, const int rounding);
VFMADDCPH __m512h _mm512_maskz_fmadd_round_pch (__mmask16 k, __m512h a, __m512h b, __m512h c, const int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E4 Class Exception Conditions.”

Additionally:

#UD If (dest_reg == src1_reg) or (dest_reg == src2_reg).

VFCMADDCSH/VFMADDCSH—Complex Multiply and Accumulate Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.MAP6.W0 57 /r VFCMADDCSH xmm1{k1}{z}, xmm2, xmm3/m32 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Complex multiply a pair of FP16 values from xmm2 and xmm3/m32, add to xmm1 and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].
EVEX.LLIG.F3.MAP6.W0 57 /r VFMADDCSH xmm1{k1}{z}, xmm2, xmm3/m32 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Complex multiply a pair of FP16 values from xmm2 and the complex conjugate of xmm3/m32, add to xmm1 and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a complex multiply and accumulate operation. There are normal and complex conjugate forms of the operation.

The masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

Operation

VFMADDCSH dest{k1}, src1, src2 (AVX512)

```
IF k1[0] or *no writemask*:
    tmp[0] := dest.fp16[0] + src1.fp16[0] * src2.fp16[0]
    tmp[1] := dest.fp16[1] + src1.fp16[1] * src2.fp16[0]

    // non-conjugate version subtracts last even term
    dest.fp16[0] := tmp[0] - src1.fp16[1] * src2.fp16[1]
    dest.fp16[1] := tmp[1] + src1.fp16[0] * src2.fp16[1]
ELSE IF *zeroing*:
    dest.fp16[0] := 0
    dest.fp16[1] := 0

DEST[127:32] := src1[127:32] // copy upper part of src1
DEST[MAXVL-1:128] := 0
```

```
VFCMADDCSH dest{k1}, src1, src2 (AVX512)
IF k1[0] or *no writemask*:
    tmp[0] := dest.fp16[0] + src1.fp16[0] * src2.fp16[0]
    tmp[1] := dest.fp16[1] + src1.fp16[1] * src2.fp16[0]

    // conjugate version subtracts odd final term
    dest.fp16[0] := tmp[0] + src1.fp16[1] * src2.fp16[1]
    dest.fp16[1] := tmp[1] - src1.fp16[0] * src2.fp16[1]
ELSE IF *zeroing*:
    dest.fp16[0] := 0
    dest.fp16[1] := 0

DEST[127:32] := src1[127:32] // copy upper part of src1
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFCMADDCSH __m128h _mm_fcmadd_round_sch (__m128h a, __m128h b, __m128h c, const int rounding);
VFCMADDCSH __m128h _mm_mask_fcmadd_round_sch (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
VFCMADDCSH __m128h _mm_mask3_fcmadd_round_sch (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
VFCMADDCSH __m128h _mm_maskz_fcmadd_round_sch (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
VFCMADDCSH __m128h _mm_fcmadd_sch (__m128h a, __m128h b, __m128h c);
VFCMADDCSH __m128h _mm_mask_fcmadd_sch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFCMADDCSH __m128h _mm_mask3_fcmadd_sch (__m128h a, __m128h b, __m128h c, __mmask8 k);
VFCMADDCSH __m128h _mm_maskz_fcmadd_sch (__mmask8 k, __m128h a, __m128h b, __m128h c);
VFCMADDCSH __m128h _mm_mask3_fmadd_round_sch (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
VFCMADDCSH __m128h _mm_mask3_fmadd_sch (__m128h a, __m128h b, __m128h c, __mmask8 k);

VFMADDCSH __m128h _mm_fmadd_round_sch (__m128h a, __m128h b, __m128h c, const int rounding);
VFMADDCSH __m128h _mm_mask_fmadd_round_sch (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
VFMADDCSH __m128h _mm_maskz_fmadd_round_sch (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
VFMADDCSH __m128h _mm_fmadd_sch (__m128h a, __m128h b, __m128h c);
VFMADDCSH __m128h _mm_mask_fmadd_sch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFMADDCSH __m128h _mm_maskz_fmadd_sch (__mmask8 k, __m128h a, __m128h b, __m128h c);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-58, “Type E10 Class Exception Conditions.”

Additionally:

#UD	If (dest_reg == src1_reg) or (dest_reg == src2_reg).

VFCMULCPH/VFMULCPH—Complex Multiply FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP6.W0 D6 /r VFCMULCPH xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Complex multiply a pair of FP16 values from xmm2 and xmm3/m128/m32bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP6.W0 D6 /r VFCMULCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Complex multiply a pair of FP16 values from ymm2 and ymm3/m256/m32bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.F2.MAP6.W0 D6 /r VFCMULCPH zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Complex multiply a pair of FP16 values from zmm2 and zmm3/m512/m32bcst, and store the result in zmm1 subject to writemask k1.
EVEX.128.F3.MAP6.W0 D6 /r VFMULCPH xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Complex multiply a pair of FP16 values from xmm2 and the complex conjugate of xmm3/m128/m32bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.F3.MAP6.W0 D6 /r VFMULCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Complex multiply a pair of FP16 values from ymm2 and the complex conjugate of ymm3/m256/m32bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.F3.MAP6.W0 D6 /r VFMULCPH zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Complex multiply a pair of FP16 values from zmm2 and the complex conjugate of zmm3/m512/m32bcst, and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a complex multiply operation. There are normal and complex conjugate forms of the operation. The broadcasting and masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

Operation

VFMULCPH dest{k1}, src1, src2 (AVX512)

```
VL = 128, 256 or 512
KL := VL/32

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF broadcasting and src2 is memory:
            tsrc2.fp16[2*i+0] := src2.fp16[0]
            tsrc2.fp16[2*i+1] := src2.fp16[1]
```

```
        ELSE:
            tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
            tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

FOR i := 0 to kl-1:
    IF k1[i] or *no writemask*:
        tmp.fp16[2*i+0] := src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
        tmp.fp16[2*i+1] := src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        // non-conjugate version subtracts last even term
        dest.fp16[2*i+0] := tmp.fp16[2*i+0] - src1.fp16[2*i+1] * tsrc2.fp16[2*i+1]
        dest.fp16[2*i+1] := tmp.fp16[2*i+1] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+1]
    ELSE IF *zeroing*:
        dest.fp16[2*i+0] := 0
        dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0
```

VFCMULCPH dest{k1}, src1, src2 (AVX512)

```
VL = 128, 256 or 512
KL := VL/32

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF broadcasting and src2 is memory:
            tsrc2.fp16[2*i+0] := src2.fp16[0]
            tsrc2.fp16[2*i+1] := src2.fp16[1]
        ELSE:
            tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
            tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        tmp.fp16[2*i+0] := src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
        tmp.fp16[2*i+1] := src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        // conjugate version subtracts odd final term
        dest.fp16[2*i] := tmp.fp16[2*i+0] +src1.fp16[2*i+1] * tsrc2.fp16[2*i+1]
        dest.fp16[2*i+1] := tmp.fp16[2*i+1] - src1.fp16[2*i+0] * tsrc2.fp16[2*i+1]
    ELSE IF *zeroing*:
        dest.fp16[2*i+0] := 0
        dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFCMULCPH __m128h _mm_cmul_pch (__m128h a, __m128h b);
VFCMULCPH __m128h _mm_mask_cmul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m128h _mm_maskz_cmul_pch (__mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m256h _mm256_cmul_pch (__m256h a, __m256h b);
VFCMULCPH __m256h _mm256_mask_cmul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m256h _mm256_maskz_cmul_pch (__mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m512h _mm512_cmul_pch (__m512h a, __m512h b);
VFCMULCPH __m512h _mm512_mask_cmul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b);
VFCMULCPH __m512h _mm512_maskz_cmul_pch (__mmask16 k, __m512h a, __m512h b);
VFCMULCPH __m512h _mm512_cmul_round_pch (__m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_mask_cmul_round_pch (__m512h src, __mmask16 k, __m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_maskz_cmul_round_pch (__mmask16 k, __m512h a, __m512h b, const int rounding);
VFCMULCPH __m128h _mm_fcmul_pch (__m128h a, __m128h b);
VFCMULCPH __m128h _mm_mask_fcmul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m128h _mm_maskz_fcmul_pch (__mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m256h _mm256_fcmul_pch (__m256h a, __m256h b);
VFCMULCPH __m256h _mm256_mask_fcmul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m256h _mm256_maskz_fcmul_pch (__mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m512h _mm512_fcmul_pch (__m512h a, __m512h b);
VFCMULCPH __m512h _mm512_mask_fcmul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b);
VFCMULCPH __m512h _mm512_maskz_fcmul_pch (__mmask16 k, __m512h a, __m512h b);
VFCMULCPH __m512h _mm512_fcmul_round_pch (__m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_mask_fcmul_round_pch (__m512h src, __mmask16 k, __m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_maskz_fcmul_round_pch (__mmask16 k, __m512h a, __m512h b, const int rounding);

VFMULCPH __m128h _mm_fmul_pch (__m128h a, __m128h b);
VFMULCPH __m128h _mm_mask_fmul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFMULCPH __m128h _mm_maskz_fmul_pch (__mmask8 k, __m128h a, __m128h b);
VFMULCPH __m256h _mm256_fmul_pch (__m256h a, __m256h b);
VFMULCPH __m256h _mm256_mask_fmul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFMULCPH __m256h _mm256_maskz_fmul_pch (__mmask8 k, __m256h a, __m256h b);
VFMULCPH __m512h _mm512_fmul_pch (__m512h a, __m512h b);
VFMULCPH __m512h _mm512_mask_fmul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b);
VFMULCPH __m512h _mm512_maskz_fmul_pch (__mmask16 k, __m512h a, __m512h b);
VFMULCPH __m512h _mm512_fmul_round_pch (__m512h a, __m512h b, const int rounding);
VFMULCPH __m512h _mm512_mask_fmul_round_pch (__m512h src, __mmask16 k, __m512h a, __m512h b, const int rounding);
VFMULCPH __m512h _mm512_maskz_fmul_round_pch (__mmask16 k, __m512h a, __m512h b, const int rounding);
VFMULCPH __m128h _mm_mask_mul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFMULCPH __m128h _mm_maskz_mul_pch (__mmask8 k, __m128h a, __m128h b);
VFMULCPH __m128h _mm_mul_pch (__m128h a, __m128h b);
VFMULCPH __m256h _mm256_mask_mul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFMULCPH __m256h _mm256_maskz_mul_pch (__mmask8 k, __m256h a, __m256h b);
VFMULCPH __m256h _mm256_mul_pch (__m256h a, __m256h b);
VFMULCPH __m512h _mm512_mask_mul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b);
VFMULCPH __m512h _mm512_maskz_mul_pch (__mmask16 k, __m512h a, __m512h b);
VFMULCPH __m512h _mm512_mul_pch (__m512h a, __m512h b);
VFMULCPH __m512h _mm512_mask_mul_round_pch (__m512h src, __mmask16 k, __m512h a, __m512h b, const int rounding);
VFMULCPH __m512h _mm512_maskz_mul_round_pch (__mmask16 k, __m512h a, __m512h b, const int rounding);
VFMULCPH __m512h _mm512_mul_round_pch (__m512h a, __m512h b, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If (dest_reg == src1_reg) or (dest_reg == src2_reg).

VFCMULCSH/VFMULCSH—Complex Multiply Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.MAP6.W0 D7 /r VFCMULCSH xmm1{k1}{z}, xmm2, xmm3/m32 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Complex multiply a pair of FP16 values from xmm2 and xmm3/m32, and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].
EVEX.LLIG.F3.MAP6.W0 D7 /r VFMULCSH xmm1{k1}{z}, xmm2, xmm3/m32 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Complex multiply a pair of FP16 values from xmm2 and the complex conjugate of xmm3/m32, and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a complex multiply operation. There are normal and complex conjugate forms of the operation. The masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

Operation

VFMULCSH dest{k1}, src1, src2 (AVX512)

```
KL := VL / 32

IF k1[0] or *no writemask*:
    // non-conjugate version subtracts last even term
    tmp.fp16[0] := src1.fp16[0] * src2.fp16[0]
    tmp.fp16[1] := src1.fp16[1] * src2.fp16[0]
    dest.fp16[0] := tmp.fp16[0] - src1.fp16[1] * src2.fp16[1]
    dest.fp16[1] := tmp.fp16[1] + src1.fp16[0] * src2.fp16[1]
ELSE IF *zeroing*:
    dest.fp16[0] := 0
    dest.fp16[1] := 0

DEST[127:32] := src1[127:32] // copy upper part of src1
DEST[MAXVL-1:128] := 0
```

```
VFCMULCSH dest{k1}, src1, src2 (AVX512)
KL := VL / 32

IF k1[0] or *no writemask*:
    tmp.fp16[0] := src1.fp16[0] * src2.fp16[0]
    tmp.fp16[1] := src1.fp16[1] * src2.fp16[0]

    // conjugate version subtracts odd final term
    dest.fp16[0] := tmp.fp16[0] + src1.fp16[1] * src2.fp16[1]
    dest.fp16[1] := tmp.fp16[1] - src1.fp16[0] * src2.fp16[1]
ELSE IF *zeroing*:
    dest.fp16[0] := 0
    dest.fp16[1] := 0

DEST[127:32] := src1[127:32] // copy upper part of src1
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFCMULCSH __m128h _mm_cmul_round_sch (__m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_mask_cmul_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_maskz_cmul_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_cmul_sch (__m128h a, __m128h b);
VFCMULCSH __m128h _mm_mask_cmul_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCSH __m128h _mm_maskz_cmul_sch (__mmask8 k, __m128h a, __m128h b);
VFCMULCSH __m128h _mm_fcmul_round_sch (__m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_mask_fcmul_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_maskz_fcmul_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_fcmul_sch (__m128h a, __m128h b);
VFCMULCSH __m128h _mm_mask_fcmul_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCSH __m128h _mm_maskz_fcmul_sch (__mmask8 k, __m128h a, __m128h b);

VFMULCSH __m128h _mm_fmul_round_sch (__m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_mask_fmul_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_maskz_fmul_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_fmul_sch (__m128h a, __m128h b);
VFMULCSH __m128h _mm_mask_fmul_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h _mm_maskz_fmul_sch (__mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h _mm_mask_mul_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_maskz_mul_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_mul_round_sch (__m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_mask_mul_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h _mm_maskz_mul_sch (__mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h _mm_mul_sch (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-58, “Type E10 Class Exception Conditions.”

Additionally:

#UD	If (dest_reg == src1_reg) or (dest_reg == src2_reg).

VFIXUPIMMPD—Fix Up Special Packed Float64 Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Perform fix-up of quad-word elements encoded in double precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each double precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in the destination with the corresponding bit clear in k1 retain their previous values or are set to 0.

Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}

FIXUPIMM_DP (dest[63:0], src1[63:0],tbl3[63:0], imm8 [7:0]){
    tsrc[63:0] := ((src1[62:52] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[63:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j := 0;
        SNAN_TOKEN: j := 1;
        ZERO_VALUE_TOKEN: j := 2;
        POS_ONE_VALUE_TOKEN: j := 3;
        NEG_INF_TOKEN: j := 4;
        POS_INF_TOKEN: j := 5;
        NEG_VALUE_TOKEN: j := 6;
        POS_VALUE_TOKEN: j := 7;
    }     ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[63:0] := dest[63:0];          ; preserve content of DEST
        0001: dest[63:0] := tsrc[63:0];          ; pass through src1 normal input value, denormal as zero
        0010: dest[63:0] := QNaN(tsrc[63:0]);
        0011: dest[63:0] := QNAN_Indefinite;
        0100: dest[63:0] := -INF;
        0101: dest[63:0] := +INF;
        0110: dest[63:0] := tsrc.sign? –INF : +INF;
        0111: dest[63:0] := -0;
        1000: dest[63:0] := +0;
        1001: dest[63:0] := -1;
        1010: dest[63:0] := +1;
        1011: dest[63:0] := ½;
        1100: dest[63:0] := 90.0;
        1101: dest[63:0] := PI/2;
        1110: dest[63:0] := MAX_FLOAT;
        1111: dest[63:0] := -MAX_FLOAT;
    }          ; end of token_response CASE

    ; The required fault reporting from imm8 is extracted
    ; TOKENs are mutually exclusive and TOKENs priority defines the order.
```

```
    ; Multiple faults related to a single token can occur simultaneously.
    IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
    IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
        ; end fault reporting
    return dest[63:0];
}       ; end of FIXUPIMM_DP()
```

VFIXUPIMMPD

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := FIXUPIMM_DP(DEST[i+63:i], SRC1[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:

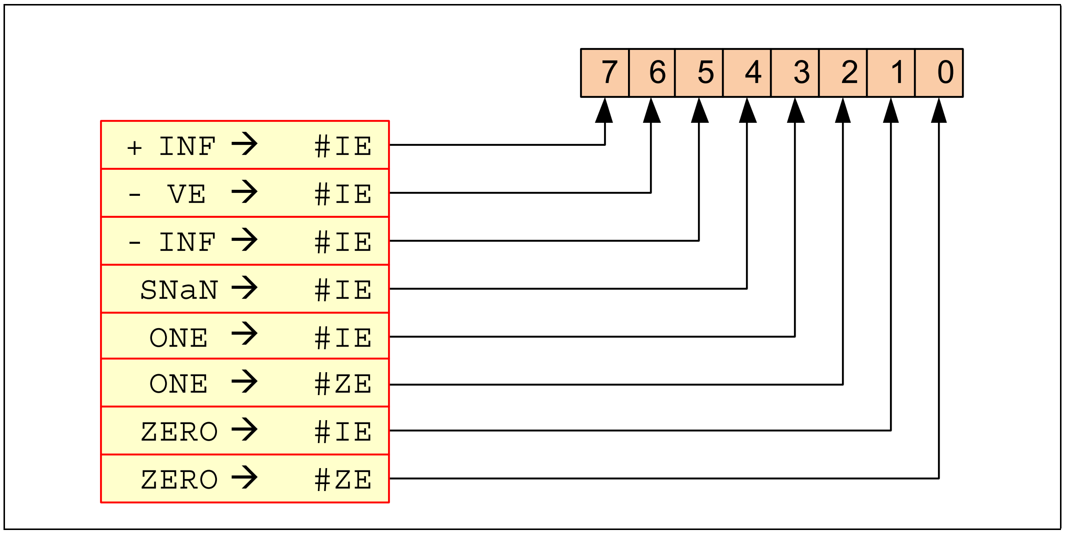


Figure 1-36. VFIXUPIMMPD Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMPD __m512d _mm512_fixupimm_pd(__m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d _mm512_mask_fixupimm_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d _mm512_maskz_fixupimm_pd(__mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d _mm512_fixupimm_round_pd(__m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m512d _mm512_mask_fixupimm_round_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m512d _mm512_maskz_fixupimm_round_pd(__mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m256d _mm256_fixupimm_pd(__m256d a, m256d b, __m256i c, int imm8);
VFIXUPIMMPD __m256d _mm256_mask_fixupimm_pd(__m256d a, __mmask8 k, __m256d b, __m256i c, int imm8);
VFIXUPIMMPD __m256d _mm256_maskz_fixupimm_pd(__mmask8 k, __m256d a, __m256d b, __m256i c, int imm8);
VFIXUPIMMPD __m128d _mm_fixupimm_pd(__m128d a, __m128d b, __m128i c, int imm8);
VFIXUPIMMPD __m128d _mm_mask_fixupimm_pd(__m128d a, __mmask8 k, __m128d b, __m128i c, int imm8);
VFIXUPIMMPD __m128d _mm_maskz_fixupimm_pd(__mmask8 k, __m128d a, __m128d b, 128ic, int imm8);

SIMD Floating-Point Exceptions

Zero, Invalid.

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”

VFIXUPIMMPS—Fix Up Special Packed Float32 Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Perform fix-up of doubleword elements encoded in single precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each single precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=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:

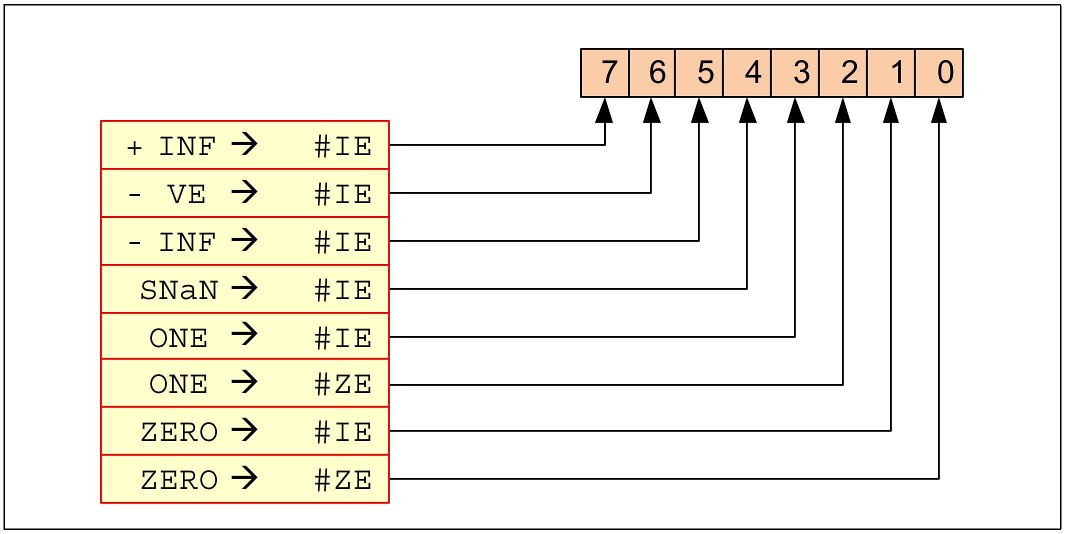


Figure 1-37. VFIXUPIMMPS Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMPS __m512 _mm512_fixupimm_ps(__m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 _mm512_mask_fixupimm_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 _mm512_maskz_fixupimm_ps(__mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 _mm512_fixupimm_round_ps(__m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 _mm512_mask_fixupimm_round_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 _mm512_maskz_fixupimm_round_ps(__mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m256 _mm256_fixupimm_ps(__m256 a, __m256 b, __m256i c, int imm8);
VFIXUPIMMPS __m256 _mm256_mask_fixupimm_ps(__m256 a, __mmask8 k, __m256 b, __m256i c, int imm8);
VFIXUPIMMPS __m256 _mm256_maskz_fixupimm_ps(__mmask8 k, __m256 a, __m256b, __m256i c, int imm8);
VFIXUPIMMPS __m128 _mm_fixupimm_ps(__m128 a, __m128 b, 128i c, int imm8);
VFIXUPIMMPS __m128 _mm_mask_fixupimm_ps(__m128 a, __mmask8 k, __m128 b, __m128i c, int imm8);
VFIXUPIMMPS __m128 _mm_maskz_fixupimm_ps(__mmask8 k, __m128 a, __m128 b, __m128i c, int imm8);

SIMD Floating-Point Exceptions

Zero, Invalid.

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”

VFIXUPIMMSD—Fix Up Special Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 55 /r ib VFIXUPIMMSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Perform a fix-up of the low quadword element encoded in double precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 64- bit memory location.

The two-level look-up table perform a fix-up of each double precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR.DAZ is used and refer to zmm2 only (i.e., zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
```

```
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}

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:

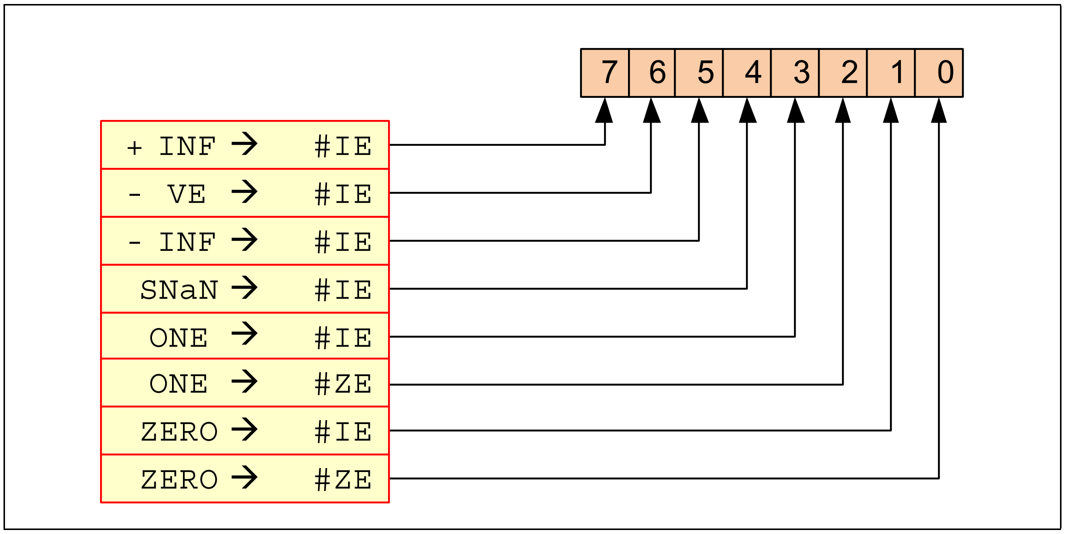


Figure 1-38. VFIXUPIMMSD Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMSD __m128d _mm_fixupimm_sd(__m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d _mm_mask_fixupimm_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d _mm_maskz_fixupimm_sd(__mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d _mm_fixupimm_round_sd(__m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d _mm_mask_fixupimm_round_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d _mm_maskz_fixupimm_round_sd(__mmask8 k, __m128d a, __m128i tbl, int imm, int sae);

SIMD Floating-Point Exceptions

Zero, Invalid.

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions.”

VFIXUPIMMSS—Fix Up Special Scalar Float32 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 55 /r ib VFIXUPIMMSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Perform a fix-up of the low doubleword element encoded in single precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low doubleword element of the destination operand (the first operand) Bits 127:32 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 32-bit memory location.

The two-level look-up table perform a fix-up of each single precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR.DAZ is used and refer to zmm2 only (i.e., zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
```

```
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}

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:

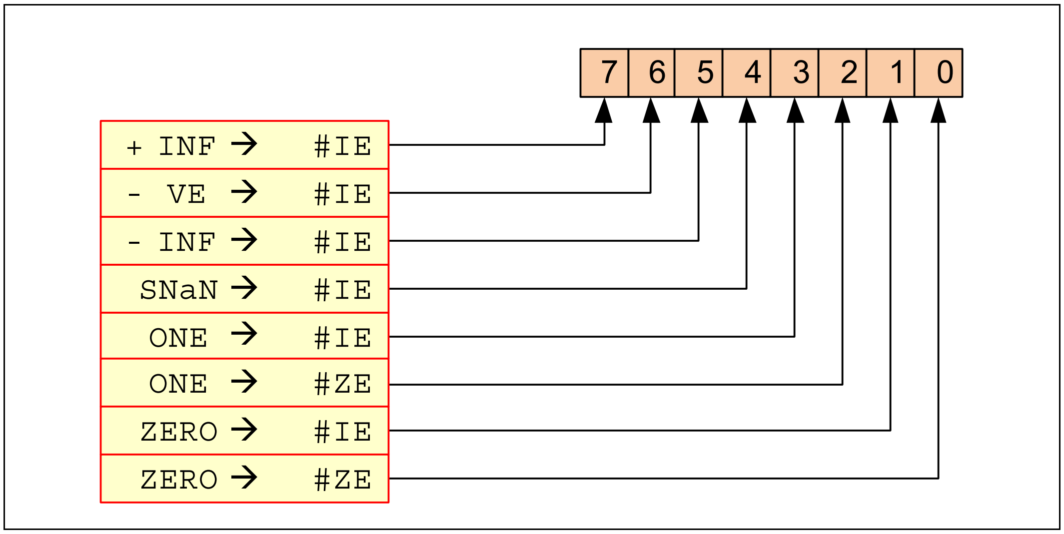


Figure 1-39. VFIXUPIMMSS Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMSS __m128 _mm_fixupimm_ss(__m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 _mm_mask_fixupimm_ss(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 _mm_maskz_fixupimm_ss(__mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 _mm_fixupimm_round_ss(__m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 _mm_mask_fixupimm_round_ss(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 _mm_maskz_fixupimm_round_ss(__mmask8 k, __m128 a, __m128i tbl, int imm, int sae);

SIMD Floating-Point Exceptions

Zero, Invalid.

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions.”

VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double Precision Floating-Point Values

Opcode/ 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	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, add to zmm1 and put result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a set of SIMD multiply-add computation on packed double precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PD: Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFMADD213PD: Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFMADD231PD: Multiplies the two, four or eight packed double precision floating-point values from the second source to the two, four or eight packed double precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] + SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD213PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] + SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD231PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] + DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMADDxxxPD __m512d _mm512_fmadd_pd(__m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_fmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask_fmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDxxxPD __m512d _mm512_mask_fmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDxxxPD __m256d _mm256_mask_fmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_maskz_fmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_mask3_fmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_mask_fmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_maskz_fmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_mask3_fmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VF[,N]MADD[132,213,231]PH—Fused Multiply-Add of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 98 /r VFMADD132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, add to xmm2, and store the result in xmm1.
EVEX.256.66.MAP6.W0 98 /r VFMADD132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, add to ymm2, and store the result in ymm1.
EVEX.512.66.MAP6.W0 98 /r VFMADD132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, add to zmm2, and store the result in zmm1.
EVEX.128.66.MAP6.W0 A8 /r VFMADD213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm1 and xmm2, add to xmm3/m128/m16bcst, and store the result in xmm1.
EVEX.256.66.MAP6.W0 A8 /r VFMADD213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm1 and ymm2, add to ymm3/m256/m16bcst, and store the result in ymm1.
EVEX.512.66.MAP6.W0 A8 /r VFMADD213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm1 and zmm2, add to zmm3/m512/m16bcst, and store the result in zmm1.
EVEX.128.66.MAP6.W0 B8 /r VFMADD231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, add to xmm1, and store the result in xmm1.
EVEX.256.66.MAP6.W0 B8 /r VFMADD231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, add to ymm1, and store the result in ymm1.
EVEX.512.66.MAP6.W0 B8 /r VFMADD231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, add to zmm1, and store the result in zmm1.
EVEX.128.66.MAP6.W0 9C /r VFNMADD132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, and negate the value. Add this value to xmm2, and store the result in xmm1.
EVEX.256.66.MAP6.W0 9C /r VFNMADD132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, and negate the value. Add this value to ymm2, and store the result in ymm1.
EVEX.512.66.MAP6.W0 9C /r VFNMADD132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, and negate the value. Add this value to zmm2, and store the result in zmm1.
EVEX.128.66.MAP6.W0 AC /r VFNMADD213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm1 and xmm2, and negate the value. Add this value to xmm3/m128/m16bcst, and store the result in xmm1.
EVEX.256.66.MAP6.W0 AC /r VFNMADD213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm1 and ymm2, and negate the value. Add this value to ymm3/m256/m16bcst, and store the result in ymm1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.MAP6.W0 AC /r VFNMADD213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm1 and zmm2, and negate the value. Add this value to zmm3/m512/m16bcst, and store the result in zmm1.
EVEX.128.66.MAP6.W0 BC /r VFNMADD231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, and negate the value. Add this value to xmm1, and store the result in xmm1.
EVEX.256.66.MAP6.W0 BC /r VFNMADD231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, and negate the value. Add this value to ymm1, and store the result in ymm1.
EVEX.512.66.MAP6.W0 BC /r VFNMADD231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, and negate the value. Add this value to zmm1, and store the result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a packed multiply-add or negated multiply-add computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The "N" (negated) forms of this instruction add the negated infinite precision intermediate product to the corresponding remaining operand. The notation' "132", "213" and "231" indicate the use of the operands in ±A * B + C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 1-8.

The destination elements are updated according to the writemask.

Table 1-8. VF[,N]MADD[132,213,231]PH Notation for Operands

Notation	Operands
132	dest = ± dest*src3+src2
231	dest = ± src2*src3+dest
213	dest = ± src2*dest+src3

Operation

VF[,N]MADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-DEST.fp16[j]*SRC3.fp16[j] + SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j]*SRC3.fp16[j] + SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-DEST.fp16[j] * t3 + SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 + SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*DEST.fp16[j] + SRC3.fp16[j])
        ELSE
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] + SRC3.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * DEST.fp16[j] + t3 )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] + t3 )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*SRC3.fp16[j] + DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*SRC3.fp16[j] + DEST.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * t3 + DEST.fp16[j] )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 + DEST.fp16[j] )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132PH, VFMADD213PH , and VFMADD231PH:
__m128h _mm_fmadd_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmadd_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmadd_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmadd_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmadd_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmadd_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmadd_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmadd_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmadd_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmadd_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmadd_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmadd_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmadd_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmadd_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmadd_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmadd_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

VFNMADD132PH, VFNMADD213PH, and VFNMADD231PH:
__m128h _mm_fnmadd_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fnmadd_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fnmadd_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fnmadd_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fnmadd_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fnmadd_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fnmadd_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fnmadd_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fnmadd_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fnmadd_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fnmadd_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fnmadd_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fnmadd_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fnmadd_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fnmadd_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fnmadd_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

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.256.66.0F38.0 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, add to zmm1 and put result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a set of SIMD multiply-add computation on packed single precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMADD213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the third source operand, performs rounding and stores the resulting the four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMADD231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 4
ELSEIF (VEX.256)
    MAXNUM := 8
```

```
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 4
ELSEIF (VEX.256)
    MAXNUM := 8
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 4
ELSEIF (VEX.256)
    MAXNUM := 8
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
```

```
            THEN DEST[i+31:i] :=
                RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
            ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                            ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
            THEN
                IF (EVEX.b = 1)
                    THEN
                        DEST[i+31:i] :=
                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                    ELSE
                        DEST[i+31:i] :=
                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                FI;
            ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                            ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
            SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
            SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
            THEN DEST[i+31:i] :=
                RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
            ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                            ; zeroing-masking
```

```
                DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMADDxxxPS __m512 _mm512_fmadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_mask_fmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_maskz_fmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_mask3_fmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDxxxPS __m512 _mm512_mask_fmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_maskz_fmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_mask3_fmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDxxxPS __m256 _mm256_mask_fmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_maskz_fmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_mask3_fmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDxxxPS __m128 _mm_mask_fmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxPS __m128 _mm_maskz_fmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 _mm_mask3_fmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxPS __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar Double Precision Floating-Point Values

Opcode/ 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.LLIG.66.0F38.W1 99 /r VFMADD132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 A9 /r VFMADD213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar double precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 B9 /r VFMADD231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD multiply-add computation on the low double precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The first and second operand are XMM registers. The third source operand can be an XMM register or a 64-bit memory location.

VFMADD132SD: Multiplies the low double precision floating-point value from the first source operand to the low double precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double precision floating-point values in the second source operand, performs rounding and stores the resulting double precision floating-point value to the destination operand (first source operand).

VFMADD213SD: Multiplies the low double precision floating-point value from the second source operand to the low double precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low double precision floating-point value in the third source operand, performs rounding and stores the resulting double precision floating-point value to the destination operand (first source operand).

VFMADD231SD: Multiplies the low double precision floating-point value from the second source to the low double precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double precision floating-point value in the first source operand, performs rounding and stores the resulting double precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN     DEST[63:0] := RoundFPControl(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMADD132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := MAXVL-1:128RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:63] := DEST[127:63]
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[127:63] := DEST[127:63]
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[127:63] := DEST[127:63]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMADDxxxSD __m128d _mm_fmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_mask_fmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_maskz_fmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_mask3_fmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxSD __m128d _mm_mask_fmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_maskz_fmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMADDxxxSD __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VF[,N]MADD[132,213,231]SH—Fused Multiply-Add of Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 99 /r VFMADD132SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm1 and xmm3/m16, add to xmm2, and store the result in xmm1.
EVEX.LLIG.66.MAP6.W0 A9 /r VFMADD213SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm1 and xmm2, add to xmm3/m16, and store the result in xmm1.
EVEX.LLIG.66.MAP6.W0 B9 /r VFMADD231SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm2 and xmm3/m16, add to xmm1, and store the result in xmm1.
EVEX.LLIG.66.MAP6.W0 9D /r VFNMADD132SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm1 and xmm3/m16, and negate the value. Add this value to xmm2, and store the result in xmm1.
EVEX.LLIG.66.MAP6.W0 AD /r VFNMADD213SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm1 and xmm2, and negate the value. Add this value to xmm3/m16, and store the result in xmm1.
EVEX.LLIG.66.MAP6.W0 BD /r VFNMADD231SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm2 and xmm3/m16, and negate the value. Add this value to xmm1, and store the result in xmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a scalar multiply-add or negated multiply-add computation on the low FP16 values using three source operands and writes the result in the destination operand. The destination operand is also the first source operand. The “N” (negated) forms of this instruction add the negated infinite precision intermediate product to the corresponding remaining operand. The notation’ “132”, “213” and “231” indicate the use of the operands in ±A * B + C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 1-9.

Bits 127:16 of the destination operand are preserved. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Table 1-9. VF[,N]MADD[132,213,231]SH Notation for Operands

Notation	Operands
132	dest = ± dest*src3+src2
231	dest = ± src2*src3+dest
213	dest = ± src2*dest+src3

Operation

VF[,N]MADD132SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form*:
        DEST.fp16[0] := RoundFPControl(-DEST.fp16[0]*SRC3.fp16[0] + SRC2.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(DEST.fp16[0]*SRC3.fp16[0] + SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

VF[,N]MADD213SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form:
        DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*DEST.fp16[0] + SRC3.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*DEST.fp16[0] + SRC3.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

VF[,N]MADD231SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form*:
        DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*SRC3.fp16[0] + DEST.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*SRC3.fp16[0] + DEST.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132SH, VFMADD213SH, and VFMADD231SH:
__m128h _mm_fmadd_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fmadd_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fmadd_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fmadd_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fmadd_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmadd_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmadd_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmadd_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);

VFNMADD132SH, VFNMADD213SH, and VFNMADD231SH:
__m128h _mm_fnmadd_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fnmadd_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fnmadd_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fnmadd_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fnmadd_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fnmadd_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fnmadd_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fnmadd_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Multiply-Add of Scalar Single Precision Floating-Point Values

Opcode/ 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.LLIG.66.0F38.W0 99 /r VFMADD132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 A9 /r VFMADD213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar single precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 B9 /r VFMADD231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD multiply-add computation on single precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The first and second operands are XMM registers. The third source operand can be a XMM register or a 32-bit memory location.

VFMADD132SS: Multiplies the low single precision floating-point value from the first source operand to the low single precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single precision floating-point value in the second source operand, performs rounding and stores the resulting single precision floating-point value to the destination operand (first source operand).

VFMADD213SS: Multiplies the low single precision floating-point value from the second source operand to the low single precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low single precision floating-point value in the third source operand, performs rounding and stores the resulting single precision floating-point value to the destination operand (first source operand).

VFMADD231SS: Multiplies the low single precision floating-point value from the second source operand to the low single precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single precision floating-point value in the first source operand, performs rounding and stores the resulting single precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
    ELSE
        IF *merging-masking*                        ; merging-masking
            THEN *DEST[31:0]] remains unchanged*
            ELSE                                    ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMADD132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMADD213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMADD231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMADDxxxSS __m128 _mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxSS __m128 _mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMADDxxxSS __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."

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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 bit Mode 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 OR AVX10.1[1]	Multiply packed double precision floating-point values from zmm1and 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 OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFMADDSUB132PD: Multiplies the two, four, or eight packed double precision floating-point values from the first source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double precision floating-point elements and subtracts the even double precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd double precision floating-point elements and subtracts the even double precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double precision floating-point elements and subtracts the even double precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMADDSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[191:128] := RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] - SRC2[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] + SRC2[255:192]
FI
```

VFMADDSUB213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
    DEST[191:128] := RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] - SRC3[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] + SRC3[255:192]
FI
```

VFMADDSUB231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[191:128] := RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] - DEST[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] + DEST[255:192]
FI
```

VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
```

```
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
                FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
                FI;
            FI

        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
```

```
                ELSE                          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
            FI
        ELSE
            IF *merging-masking*              ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
```

```
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                        ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                    ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            FI
        ELSE
            IF *merging-masking*                        ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                    ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
```

```
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[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 Table 2-19, "Type 2 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."

VFMADDSUB132PH/VFMADDSUB213PH/VFMADDSUB231PH—Fused Multiply-Alternating Add/Subtract of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 96 /r VFMADDSUB132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, add/subtract elements in xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 96 /r VFMADDSUB132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, add/subtract elements in ymm2, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 96 /r VFMADDSUB132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, add/subtract elements in zmm2, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 A6 /r VFMADDSUB213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 A6 /r VFMADDSUB213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 A6 /r VFMADDSUB213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 B6 /r VFMADDSUB231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, add/subtract elements in xmm1, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 B6 /r VFMADDSUB231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, add/subtract elements in ymm1, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 B6 /r VFMADDSUB231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, add/subtract elements in zmm1, and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a packed multiply-add (odd elements) or multiply-subtract (even elements) computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The notation' "132", "213" and "231" indicate the use of the operands in A * B ± C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 1-13.

The destination elements are updated according to the writemask.

Table 1-10. VFMADDSUB[132,213,231]PH Notation for Odd and Even Elements

Notation	Odd Elements	Even Elements
132	dest = dest*src3+src2	dest = dest*src3-src2
231	dest = src2*src3+dest	dest = src2*src3-dest
213	dest = src2*dest+src3	dest = src2*dest-src3

Operation

VFMADDSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * SRC3.fp16[j] - SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * SRC3.fp16[j] + SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMADDSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 - SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 + SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
```

```
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] - SRC3.fp16[j])
        ELSE
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] + SRC3.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] - t3)
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] + t3)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * SRC3.fp16[j] - DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * SRC3.fp16[j] + DEST.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 - DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 + DEST.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUB132PH, VFMADDSUB213PH, and VFMADDSUB231PH:
__m128h _mm_fmaddsub_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmaddsub_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmaddsub_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmaddsub_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmaddsub_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmaddsub_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmaddsub_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmaddsub_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmaddsub_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmaddsub_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmaddsub_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmaddsub_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmaddsub_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmaddsub_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmaddsub_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmaddsub_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single Precision Floating-Point Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 A6 /r VFMADDSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFMADDSUB132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single precision floating-point elements and subtracts the even single precision floating-point values in the second source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd single precision floating-point elements and subtracts the even single precision floating-point values in the third source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single precision floating-point elements and subtracts the even single precision floating-point values in the first source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADDSUB132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM :=2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] + SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADDSUB213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] + SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADDSUB231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
    DEST[n+63:n+32] :=RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] + DEST[n+63:n+32])
}
```

```
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) (4, 128), (8, 256),= (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
```

```
                        DEST[i+31:i] :=
                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                    ELSE
                        DEST[i+31:i] :=
                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                FI;
            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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
```

```
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX Encoded Version, When SRC3 Operand is a Register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
```

```
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX Encoded Version, When SRC3 Operand is a Memory Source)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[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

```
VFMADDSUBxxxPS __m512 _mm512_fmaddsub_ps(__m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 _mm512_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 _mm512_mask_fmaddsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 _mm512_maskz_fmaddsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 _mm512_mask3_fmaddsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDSUBxxxPS __m512 _mm512_mask_fmaddsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 _mm512_maskz_fmaddsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 _mm512_mask3_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDSUBxxxPS __m256 _mm256_mask_fmaddsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 _mm256_maskz_fmaddsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 _mm256_mask3_fmaddsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDSUBxxxPS __m128 _mm_mask_fmaddsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 _mm_maskz_fmaddsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
```

VFMADDSUBxxxPS __m128 _mm_mask3_fmaddsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDSUBxxxPS __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double Precision Floating-Point Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract 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 9A /r VFMSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract zmm1 and put result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a set of SIMD multiply-subtract computation on packed double precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PD: Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFMSUB213PD: Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFMSUB231PD: Multiplies the two, four or eight packed double precision floating-point values from the second source to the two, four or eight packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132PD DEST, SRC2, SRC3 (VEX Encoded Versions)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB213PD DEST, SRC2, SRC3 (VEX Encoded Versions)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB231PD DEST, SRC2, SRC3 (VEX Encoded Versions)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] - DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
```

```
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB132PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB132PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Memory Source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
            FI;
            ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB213PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
```

```
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB213PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Memory Source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
+31:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB231PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
```

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB231PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Memory Source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMSUBxxxPD __m512d _mm512_fmsub_pd(__m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_fmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_mask_fmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_mask3_fmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBxxxPD __m512d _mm512_mask_fmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_mask3_fmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBxxxPD __m256d _mm256_mask_fmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_maskz_fmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_mask3_fmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm_mask_fmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm_maskz_fmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
```

VFMSUBxxxPD __m128d _mm_mask3_fmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VF[,N]MSUB[132,213,231]PH—Fused Multiply-Subtract of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 9A /r VFMSUB132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, subtract xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 9A /r VFMSUB132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, subtract ymm2, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 9A /r VFMSUB132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, subtract zmm2, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 AA /r VFMSUB213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm1 and xmm2, subtract xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 AA /r VFMSUB213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm1 and ymm2, subtract ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 AA /r VFMSUB213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm1 and zmm2, subtract zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 BA /r VFMSUB231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, subtract xmm1, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 BA /r VFMSUB231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, subtract ymm1, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 BA /r VFMSUB231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, subtract zmm1, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 9E /r VFNMSUB132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, and negate the value. Subtract xmm2 from this value, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 9E /r VFNMSUB132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, and negate the value. Subtract ymm2 from this value, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 9E /r VFNMSUB132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, and negate the value. Subtract zmm2 from this value, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 AE /r VFNMSUB213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm1 and xmm2, and negate the value. Subtract xmm3/m128/m16bcst from this value, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 AE /r VFNMSUB213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm1 and ymm2, and negate the value. Subtract ymm3/m256/m16bcst from this value, and store the result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.MAP6.W0 AE /r VFNMSUB213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm1 and zmm2, and negate the value. Subtract zmm3/m512/m16bcst from this value, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 BE /r VFNMSUB231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, and negate the value. Subtract xmm1 from this value, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 BE /r VFNMSUB231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, and negate the value. Subtract ymm1 from this value, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 BE /r VFNMSUB231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, and negate the value. Subtract zmm1 from this value, and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a packed multiply-subtract or a negated multiply-subtract computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The “N” (negated) forms of this instruction subtract the remaining operand from the negated infinite precision intermediate product. The notation’ “132”, “213” and “231” indicate the use of the operands in ±A * B ? C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 1-11.

The destination elements are updated according to the writemask.

Table 1-11. VF[,N]MSUB[132,213,231]PH Notation for Operands

Notation	Operands
132	dest = ± dest*src3-src2
231	dest = ± src2*src3-dest
213	dest = ± src2*dest-src3

Operation

VF[,N]MSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-DEST.fp16[j]*SRC3.fp16[j] - SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j]*SRC3.fp16[j] - SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-DEST.fp16[j] * t3 - SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 - SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*DEST.fp16[j] - SRC3.fp16[j])
        ELSE
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] - SRC3.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * DEST.fp16[j] - t3 )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] - t3 )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*SRC3.fp16[j] - DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*SRC3.fp16[j] - DEST.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * t3 - DEST.fp16[j] )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 - DEST.fp16[j] )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132PH, VFMSUB213PH, and VFMSUB231PH:
__m128h _mm_fmsub_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmsub_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmsub_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmsub_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmsub_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmsub_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmsub_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmsub_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmsub_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmsub_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmsub_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmsub_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmsub_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmsub_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmsub_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmsub_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

VFNMSUB132PH, VFNMSUB213PH, and VFNMSUB231PH:
__m128h _mm_fnmsub_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fnmsub_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fnmsub_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fnmsub_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fnmsub_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fnmsub_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fnmsub_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fnmsub_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fnmsub_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fnmsub_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fnmsub_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fnmsub_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fnmsub_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fnmsub_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fnmsub_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fnmsub_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/En	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.256.66.0F38.0 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract zmm1 and put result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a set of SIMD multiply-subtract computation on packed single precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMSUB213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMSUB231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source to the four, eight or sixteen packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
```

```
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
```

```
            THEN DEST[i+31:i] :=
                RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                                ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
```

```
                DEST[i+31:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[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 Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double Precision Floating-Point Values

Opcode/ 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.LLIG.66.0F38.W1 9B /r VFMSUB132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 AB /r VFMSUB213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar double precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 BB /r VFMSUB231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD multiply-subtract computation on the low packed double precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 64-bit memory location.

VFMSUB132SD: Multiplies the low packed double precision floating-point value from the first source operand to the low packed double precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double precision floating-point values in the second source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFMSUB213SD: Multiplies the low packed double precision floating-point value from the second source operand to the low packed double precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed double precision floating-point value in the third source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFMSUB231SD: Multiplies the low packed double precision floating-point value from the second source to the low packed double precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double precision floating-point value in the first source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMSUBxxxSD __m128d _mm_fmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask_fmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxSD __m128d _mm_mask_fmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMSUBxxxSD __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VF[,N]MSUB[132,213,231]SH—Fused Multiply-Subtract of Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 9B /r VFMSUB132SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm1 and xmm3/m16, subtract xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 AB /r VFMSUB213SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm1 and xmm2, subtract xmm3/m16, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 BB /r VFMSUB231SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm2 and xmm3/m16, subtract xmm1, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 9F /r VFNMSUB132SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm1 and xmm3/m16, and negate the value. Subtract xmm2 from this value, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 AF /r VFNMSUB213SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm1 and xmm2, and negate the value. Subtract xmm3/m16 from this value, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 BF /r VFNMSUB231SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm2 and xmm3/m16, and negate the value. Subtract xmm1 from this value, and store the result in xmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a scalar multiply-subtract or negated multiply-subtract computation on the low FP16 values using three source operands and writes the result in the destination operand. The destination operand is also the first source operand. The “N” (negated) forms of this instruction subtract the remaining operand from the negated infinite precision intermediate product. The notation’ “132”, “213” and “231” indicate the use of the operands in ±A * B − C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 1-14.

Bits 127:16 of the destination operand are preserved. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Table 1-12. VF[,N]MSUB[132,213,231]SH Notation for Operands

Notation	Operands
132	dest = ± dest*src3-src2
231	dest = ± src2*src3-dest
213	dest = ± src2*dest-src3

Operation

VF[,N]MSUB132SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form*:
        DEST.fp16[0] := RoundFPControl(-DEST.fp16[0]*SRC3.fp16[0] - SRC2.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(DEST.fp16[0]*SRC3.fp16[0] - SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

VF[,N]MSUB213SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form:
        DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*DEST.fp16[0] - SRC3.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*DEST.fp16[0] - SRC3.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

VF[,N]MSUB231SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form*:
        DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*SRC3.fp16[0] - DEST.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*SRC3.fp16[0] - DEST.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132SH, VFMSUB213SH, and VFMSUB231SH:
__m128h _mm_fmsub_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fmsub_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fmsub_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fmsub_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fmsub_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmsub_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmsub_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmsub_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);

VFNMSUB132SH, VFNMSUB213SH, and VFNMSUB231SH:
__m128h _mm_fnmsub_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fnmsub_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fnmsub_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fnmsub_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fnmsub_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fnmsub_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fnmsub_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fnmsub_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single Precision Floating-Point Values

Opcode/ 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.LLIG.66.0F38.W0 9B /r VFMSUB132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 AB /r VFMSUB213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar single precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 BB /r VFMSUB231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD multiply-subtract computation on the low packed single precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 32-bit memory location.

VFMSUB132SS: Multiplies the low packed single precision floating-point value from the first source operand to the low packed single precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single precision floating-point values in the second source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VFMSUB213SS: Multiplies the low packed single precision floating-point value from the second source operand to the low packed single precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed single precision floating-point value in the third source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VFMSUB231SS: Multiplies the low packed single precision floating-point value from the second source to the low packed single precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single precision floating-point value in the first source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(SRC2[31:0]*SRC3[63:0] - DEST[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] - DEST[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMSUBxxxSS __m128 _mm_fmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask_fmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxSS __m128 _mm_mask_fmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMSUBxxxSS __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double Precision Floating-Point Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFMSUBADD132PD: Multiplies the two, four, or eight packed double precision floating-point values from the first source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double precision floating-point elements and adds the even double precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd double precision floating-point elements and adds the even double precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double precision floating-point elements and adds the even double precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a

XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[191:128] := RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] + SRC2[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] - SRC2[255:192]
FI
```

VFMSUBADD213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
    DEST[191:128] := RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] + SRC3[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] - SRC3[255:192]
FI
```

VFMSUBADD231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[191:128] := RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] + DEST[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] - DEST[255:192]
FI
```

VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
```

```
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
                    FI;
            FI

        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
```

```
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
```

```
                        RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
                    ELSE
                        DEST[i+63:i] :=
                        RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                        FI;
                FI
            ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE                            ; zeroing-masking
                        DEST[i+63:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
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 Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VFMSUBADD132PH/VFMSUBADD213PH/VFMSUBADD231PH—Fused Multiply-Alternating Subtract/Add of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 97 /r VFMSUBADD132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, subtract/add elements in xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 97 /r VFMSUBADD132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, subtract/add elements in ymm2, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 97 /r VFMSUBADD132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, subtract/add elements in zmm2, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 A7 /r VFMSUBADD213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 A7 /r VFMSUBADD213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 A7 /r VFMSUBADD213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 B7 /r VFMSUBADD231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, subtract/add elements in xmm1, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 B7 /r VFMSUBADD231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, subtract/add elements in ymm1, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 B7 /r VFMSUBADD231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, subtract/add elements in zmm1, and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a packed multiply-add (even elements) or multiply-subtract (odd elements) computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The notation "132", "213" and "231" indicate the use of the operands in A * B ± C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 1-13.

The destination elements are updated according to the writemask.

Table 1-13. VFMSUBADD[132,213,231]PH Notation for Odd and Even Elements

Notation	Odd Elements	Even Elements
132	dest = dest*src3-src2	dest = dest*src3+src2
231	dest = src2*src3-dest	dest = src2*src3+dest
213	dest = src2*dest-src3	dest = src2*dest+src3

Operation

VFMSUBADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j]*SRC3.fp16[j] + SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j]*SRC3.fp16[j] - SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMSUBADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 + SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 - SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
```

```
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0:
```

VFMSUBADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] + SRC3.fp16[j])
        ELSE
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] - SRC3.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMSUBADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] + t3 )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] - t3 )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0:
```

VFMSUBADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*SRC3.fp16[j] + DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*SRC3.fp16[j] - DEST.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMSUBADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 + DEST.fp16[j] )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 - DEST.fp16[j] )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADD132PH, VFMSUBADD213PH, and VFMSUBADD231PH:
__m128h _mm_fmsubadd_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmsubadd_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmsubadd_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmsubadd_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmsubadd_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmsubadd_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmsubadd_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmsubadd_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmsubadd_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmsubadd_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmsubadd_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmsubadd_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmsubadd_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmsubadd_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmsubadd_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmsubadd_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single Precision Floating-Point Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EVEX.512.66.0F38.W0 97 /r VFMSUBADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFMSUBADD132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single precision floating-point elements and adds the even single precision floating-point values in the second source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single precision floating-point elements and adds the even single precision floating-point values in the third source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single precision floating-point elements and adds the even single precision floating-point values in the first source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] -SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUBADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] +SRC3[n+31:n])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] -SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUBADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] -DEST[n+63:n+32])
}
```

```
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                        RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                        RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
```

```
                    DEST[i+31:i] :=
                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                  ELSE
                    DEST[i+31:i] :=
                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
                FI;
            FI

        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
```

```
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                    ELSE
                        DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
                    FI;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
```

```
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                        RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                        RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                    FI;
            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 Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VFNMADD132PD/VFNMADD213PD/VFNMADD231PD—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 VFNMADD132PD 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 VFNMADD213PD 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 VFNMADD231PD 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 VFNMADD132PD 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 VFNMADD213PD 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 VFNMADD231PD 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 VFNMADD132PD xmm0 {k1}{z}, xmm1, xmm2/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm1 and put result in ymm1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 9C /r VFNMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	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 VFNMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	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 VFNMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMADD132PD: Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMADD213PD: Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMADD231PD: Multiplies the two, four or eight packed double precision floating-point values from the second source to the two, four or eight packed double precision floating-point values in the third source operand, the negated infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMADD132PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(-(DEST[n+63:n]*SRC3[n+63:n]) + SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD213PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(-(SRC2[n+63:n]*DEST[n+63:n]) + SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD231PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(-(SRC2[n+63:n]*SRC3[n+63:n]) + DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
```

```
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) + SRC2[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[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
```

VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
```

```
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[63:0])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
```

```
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[63:0]) + DEST[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                  ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADDxxxPD __m512d _mm512_fnmadd_pd(__m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMADDxxxPD __m256d _mm256_mask_fnmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_maskz_fnmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_mask3_fnmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_mask_fnmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_maskz_fnmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_mask3_fnmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VFNMADD132PS/VFNMADD213PS/VFNMADD231PS—Fused Negative Multiply-Add of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 9C /r VFNMADD132PS 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 VFNMADD213PS 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 VFNMADD231PS 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 VFNMADD132PS 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 VFNMADD213PS 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 VFNMADD231PS 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 VFNMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 9C /r VFNMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	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 VFNMADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMADD132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFNMADD213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the third source operand, performs rounding and stores the resulting the four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFNMADD231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFNMADD132PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(- (DEST[n+31:n]*SRC3[n+31:n]) + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD213PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(- (SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD231PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(- (SRC2[n+31:n]*SRC3[n+31:n]) + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
```

```
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[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
```

VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
```

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[31:0])

                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
```

```
            ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                            ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) + DEST[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADDxxxPS __m512 _mm512_fnmadd_ps(__m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_mask_fnmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_maskz_fnmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_mask3_fnmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMADDxxxPS __m512 _mm512_mask_fnmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_maskz_fnmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_mask3_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFNMADDxxxPS __m256 _mm256_mask_fnmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_maskz_fnmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_mask3_fnmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMADDxxxPS __m128 _mm_mask_fnmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm_maskz_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm_mask3_fnmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxPS __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VFNMADD132SD/VFNMADD213SD/VFNMADD231SD—Fused Negative Multiply-Add of Scalar Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W1 9D /r VFNMADD132SD 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 VFNMADD213SD 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 VFNMADD231SD 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.LLIG.66.0F38.W1 9D /r VFNMADD132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 AD /r VFNMADD213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m64 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 BD /r VFNMADD231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMADD132SD: Multiplies the low packed double precision floating-point value from the first source operand to the low packed double precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double precision floating-point values in the second source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFNMADD213SD: Multiplies the low packed double precision floating-point value from the second source operand to the low packed double precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed double precision floating-point value in the third source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFNMADD231SD: Multiplies the low packed double precision floating-point value from the second source to the low packed double precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double precision floating-point value in the first source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFNMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMADD132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMADD213SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMADD231SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFNMADDxxxSD __m128d _mm_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_mask_fnmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMADDxxxSD __m128d _mm_maskz_fnmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxSD __m128d _mm_mask_fnmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_maskz_fnmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMADDxxxSD __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VFNMADD132SS/VFNMADD213SS/VFNMADD231SS—Fused Negative Multiply-Add of Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W0 9D /r VFNMADD132SS 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 VFNMADD213SS 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 VFNMADD231SS 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.LLIG.66.0F38.W0 9D /r VFNMADD132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 AD /r VFNMADD213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 BD /r VFNMADD231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFNMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) + DEST[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMADD132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMADD213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMADD231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) + DEST[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFNMADDxxxSS __m128 _mm_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_mask_fnmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxSS __m128 _mm_maskz_fnmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxSS __m128 _mm_mask3_fnmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxSS __m128 _mm_mask_fnmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_maskz_fnmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_mask3_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMADDxxxSS __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD—Fused Negative Multiply-Subtract of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 9E /r VFNMSUB132PD 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 VFNMSUB213PD 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 VFNMSUB231PD 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 VFNMSUB132PD 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 VFNMSUB213PD 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 VFNMSUB231PD 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 VFNMSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm1 and put result in ymm1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 9E /r VFNMSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	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 VFNMSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	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 VFNMSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMSUB132PD: Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMSUB213PD: Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMSUB231PD: Multiplies the two, four or eight packed double precision floating-point values from the second source to the two, four or eight packed double precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR( - (DEST[n+63:n]*SRC3[n+63:n]) - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB213PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR( - (SRC2[n+63:n]*DEST[n+63:n]) - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB231PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR( - (SRC2[n+63:n]*SRC3[n+63:n]) - DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
```

```
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[63:0]) - SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
```

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[63:0])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
        ELSE
```

```
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE                            ; zeroing-masking
                        DEST[i+63:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[63:0]) - DEST[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

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 Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS—Fused Negative Multiply-Subtract of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 9E /r VFNMSUB132PS 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 VFNMSUB213PS 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 VFNMSUB231PS 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 VFNMSUB132PS 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 VFNMSUB213PS 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.256.66.0F38.0 BE /r VFNMSUB231PS 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 VFNMSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 VFNMSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 9E /r VFNMSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	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 VFNMSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	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 VFNMSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMSUB132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
```

```
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
```

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[31:0])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
        ELSE
```

```
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                                ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) - DEST[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[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 Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD—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 VFNMSUB132SD 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 VFNMSUB213SD 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 VFNMSUB231SD 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.LLIG.66.0F38.W1 9F /r VFNMSUB132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 AF /r VFNMSUB213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m64 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 BF /r VFNMSUB231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and subtract xmm1 and put result in xmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMSUB132SD: Multiplies the low packed double precision floating-point value from the first source operand to the low packed double precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double precision floating-point value in the second source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFNMSUB213SD: Multiplies the low packed double precision floating-point value from the second source operand to the low packed double precision floating-point value in the first source operand. From negated infinite precision intermediate result, subtracts the low double precision floating-point value in the third source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFNMSUB231SD: Multiplies the low packed double precision floating-point value from the second source to the low packed double precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double precision floating-point value in the first source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFNMSUBxxxSD __m128d _mm_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_mask_fnmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d _mm_maskz_fnmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d _mm_mask3_fnmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxxSD __m128d _mm_mask_fnmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_maskz_fnmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_mask3_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMSUBxxxSD __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VF[,N]MSUB[132,213,231]SH—Fused Multiply-Subtract of Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 9B /r VFMSUB132SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm1 and xmm3/m16, subtract xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 AB /r VFMSUB213SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm1 and xmm2, subtract xmm3/m16, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 BB /r VFMSUB231SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm2 and xmm3/m16, subtract xmm1, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 9F /r VFNMSUB132SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm1 and xmm3/m16, and negate the value. Subtract xmm2 from this value, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 AF /r VFNMSUB213SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm1 and xmm2, and negate the value. Subtract xmm3/m16 from this value, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 BF /r VFNMSUB231SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply FP16 values from xmm2 and xmm3/m16, and negate the value. Subtract xmm1 from this value, and store the result in xmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a scalar multiply-subtract or negated multiply-subtract computation on the low FP16 values using three source operands and writes the result in the destination operand. The destination operand is also the first source operand. The “N” (negated) forms of this instruction subtract the remaining operand from the negated infinite precision intermediate product. The notation’ “132”, “213” and “231” indicate the use of the operands in ±A * B − C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 1-14.

Bits 127:16 of the destination operand are preserved. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Table 1-14. VF[,N]MSUB[132,213,231]SH Notation for Operands

Notation	Operands
132	dest = ± dest*src3-src2
231	dest = ± src2*src3-dest
213	dest = ± src2*dest-src3

Operation

VF[,N]MSUB132SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form*:
        DEST.fp16[0] := RoundFPControl(-DEST.fp16[0]*SRC3.fp16[0] - SRC2.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(DEST.fp16[0]*SRC3.fp16[0] - SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

VF[,N]MSUB213SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form:
        DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*DEST.fp16[0] - SRC3.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*DEST.fp16[0] - SRC3.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

VF[,N]MSUB231SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form*:
        DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*SRC3.fp16[0] - DEST.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*SRC3.fp16[0] - DEST.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132SH, VFMSUB213SH, and VFMSUB231SH:

```
__m128h _mm_fmsub_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fmsub_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fmsub_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fmsub_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fmsub_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmsub_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmsub_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmsub_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);
```

VFNMSUB132SH, VFNMSUB213SH, and VFNMSUB231SH:

```
__m128h _mm_fnmsub_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fnmsub_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fnmsub_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fnmsub_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fnmsub_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fnmsub_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fnmsub_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fnmsub_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS—Fused Negative Multiply-Subtract of Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W0 9F /r VFNMSUB132SS 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 VFNMSUB213SS 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 VFNMSUB231SS 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.LLIG.66.0F38.W0 9F /r VFNMSUB132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 AF /r VFNMSUB213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 BF /r VFNMSUB231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1[1]	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) - DEST[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) - DEST[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFNMSUBxxxSS __m128 _mm_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_mask_fnmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 _mm_maskz_fnmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 _mm_mask3_fnmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxSS __m128 _mm_mask_fnmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_maskz_fnmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMSUBxxxSS __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VFPCLASSPD—Tests Types of Packed Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, xmm2/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

The FPCLASSPD instruction checks the packed double precision floating-point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX_KL-1:8/4/2] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 1-40. The classification test for each category is listed in Table 1-11.

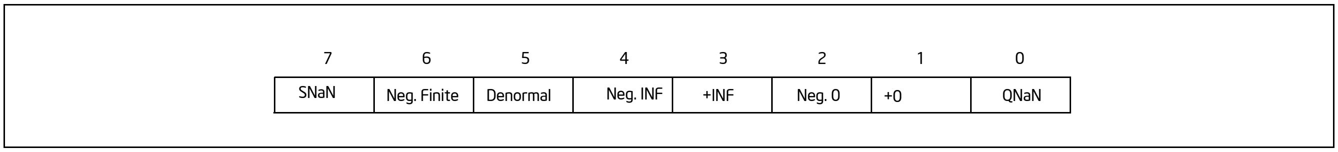


Figure 1-40. Imm8 Byte Specifier of Special Case Floating-Point Values for VFPCLASSPD/SD/PS/SS

Table 1-11. 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
    PInf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res := ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
            ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
            ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
            ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} //* end of CheckFPClassDP() *//
```

VFPCLASSPD (EVEX Encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN
                    DEST[j] := CheckFPClassDP(SRC1[63:0], imm8[7:0]);
                ELSE
                    DEST[j] := CheckFPClassDP(SRC1[i+63:i], imm8[7:0]);
```

```
                FI;
            ELSE DEST[j] := 0                       ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSPD __mmask8 _mm512_fpclass_pd_mask(__m512d a, int c);
VFPCLASSPD __mmask8 _mm512_mask_fpclass_pd_mask(__mmask8 m, __m512d a, int c)
VFPCLASSPD __mmask8 _mm256_fpclass_pd_mask(__m256d a, int c)
VFPCLASSPD __mmask8 _mm256_mask_fpclass_pd_mask(__mmask8 m, __m256d a, int c)
VFPCLASSPD __mmask8 _mm_fpclass_pd_mask(__m128d a, int c)
VFPCLASSPD __mmask8 _mm_mask_fpclass_pd_mask(__mmask8 m, __m128d a, int c)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VFPCLASSPH—Test Types of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.0F3A.W0 66 /r /ib VFPCLASSPH k1{k2}, xmm1/m128/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.256.NP.0F3A.W0 66 /r /ib VFPCLASSPH k1{k2}, ymm1/m256/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.512.NP.0F3A.W0 66 /r /ib VFPCLASSPH k1{k2}, zmm1/m512/m16bcst, imm8	A	V/V	AVX512-FP16 OR AVX10.1[1]	Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8 (r)	N/A

Description

This instruction checks the packed FP16 values in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against; see Table 1-15 for the categories. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the corresponding bits in the destination mask register according to the writemask.

Table 1-15. Classifier Operations for VFPCLASSPH/VFPCLASSSH

Bits	Category	Classifier
imm8[0]	QNAN	Checks for QNAN
imm8[1]	PosZero	Checks +0
imm8[2]	NegZero	Checks for -0
imm8[3]	PosINF	Checks for +∞
imm8[4]	NegINF	Checks for −∞
imm8[5]	Denormal	Checks for Denormal
imm8[6]	Negative	Checks for Negative finite
imm8[7]	SNAN	Checks for SNAN

Operation

```
def check_fp_class_fp16(tsrc, imm8):
    negative := tsrc[15]
    exponent_all_ones := (tsrc[14:10] == 0x1F)
    exponent_all_zeros := (tsrc[14:10] == 0)
    mantissa_all_zeros := (tsrc[9:0] == 0)
    zero := exponent_all_zeros and mantissa_all_zeros
    signaling_bit := tsrc[9]

    snan := exponent_all_ones and not(mantissa_all_zeros) and not(signaling_bit)
    qnan := exponent_all_ones and not(mantissa_all_zeros) and signaling_bit
    positive_zero := not(negative) and zero
    negative_zero := negative and zero
    positive_infinity := not(negative) and exponent_all_ones and mantissa_all_zeros
    negative_infinity := negative and exponent_all_ones and mantissa_all_zeros
    denormal := exponent_all_zeros and not(mantissa_all_zeros)
    finite_negative := negative and not(exponent_all_ones) and not(zero)

    return (imm8[0] and qnan) OR
        (imm8[1] and positive_zero) OR
        (imm8[2] and negative_zero) OR
        (imm8[3] and positive_infinity) OR
        (imm8[4] and negative_infinity) OR
        (imm8[5] and denormal) OR
        (imm8[6] and finite_negative) OR
        (imm8[7] and snan)
```

VFPCLASSPH dest{k2}, src, imm8

```
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k2[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := SRC.fp16[0]
        ELSE:
            tsrc := SRC.fp16[i]
        DEST.bit[i] := check_fp_class_fp16(tsrc, imm8)
    ELSE:
        DEST.bit[i] := 0

DEST[MAXKL-1:kl] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFPCLASSPH __mmask8 _mm_fpclass_ph_mask (__m128h a, int imm8);
VFPCLASSPH __mmask8 _mm_mask_fpclass_ph_mask (__mmask8 k1, __m128h a, int imm8);
VFPCLASSPH __mmask16 _mm256_fpclass_ph_mask (__m256h a, int imm8);
VFPCLASSPH __mmask16 _mm256_mask_fpclass_ph_mask (__mmask16 k1, __m256h a, int imm8);
VFPCLASSPH __mmask32 _mm512_fpclass_ph_mask (__m512h a, int imm8);
VFPCLASSPH __mmask32 _mm512_mask_fpclass_ph_mask (__mmask32 k1, __m512h a, int imm8);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E4 Class Exception Conditions.”

VFPCLASSPS—Tests Types of 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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

The 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 1-40. The classification test for each category is listed in Table 1-11.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
CheckFPClassSP (tsrc[31:0], imm8[7:0]){

    //* Start checking the source operand for special type *//
    NegNum  := tsrc[31];
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes := 1; FI;
    IF (tsrc[30:23]=0h) Then ExpAllZeros := 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros := 1;
    ELSIF (tsrc[22:0]=0h) Then
```

```
        MantAllZeros := 1;
    FI;
    ZeroNumber= ExpAllZeros AND MantAllZeros
    SignalingBit= tsrc[22];

    sNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res := ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
                ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
                ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
                ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} //* end of CheckSPClassSP() *//
```

VFPCLASSPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN
                    DEST[j] := CheckFPClassDP(SRC1[31:0], imm8[7:0]);
                ELSE
                    DEST[j] := CheckFPClassDP(SRC1[i+31:i], imm8[7:0]);
            FI;
        ELSE  DEST[j] := 0                    ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSPS __mmask16 _mm512_fpclass_ps_mask(__m512 a, int c);
VFPCLASSPS __mmask16 _mm512_mask_fpclass_ps_mask(__mmask16 m, __m512 a, int c)
VFPCLASSPS __mmask8 _mm256_fpclass_ps_mask(__m256 a, int c)
VFPCLASSPS __mmask8 _mm256_mask_fpclass_ps_mask(__mmask8 m, __m256 a, int c)
VFPCLASSPS __mmask8 _mm_fpclass_ps_mask(__m128 a, int c)
VFPCLASSPS __mmask8 _mm_mask_fpclass_ps_mask(__mmask8 m, __m128 a, int c)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VFPCLASSSD—Tests Type of a Scalar Float64 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 67 /r ib VFPCLASSSD k2 {k1}, xmm2/m64, imm8	A	V/V	AVX512DQ OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

The FPCLASSSD instruction checks the low double precision floating-point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX_KL-1: 1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 1-40. The classification test for each category is listed in Table 1-11.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
CheckFPClassDP (tsrc[63:0], imm8[7:0]){

    NegNum  := tsrc[63];
    IF (tsrc[62:52]=07FFh) Then ExpAllOnes := 1; FI;
    IF (tsrc[62:52]=0h) Then ExpAllZeros := 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros := 1;
    ELSIF (tsrc[51:0]=0h) Then
        MantAllZeros := 1;
    FI;
    ZeroNumber := ExpAllZeros AND MantAllZeros
    SignalingBit := tsrc[51];

    sNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res := ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite
```

```
    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
                ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
                ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
                ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} //* end of CheckFPClassDP() *//
```

VFPCLASSSD (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[0] :=
                CheckFPClassDP(SRC1[63:0], imm8[7:0])
    ELSE  DEST[0] := 0                        ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSSD __mmask8 _mm_fpclass_sd_mask(__m128d a, int c)
VFPCLASSSD __mmask8 _mm_mask_fpclass_sd_mask(__mmask8 m, __m128d a, int c)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-53, “Type E6 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VFPCLASSSH—Test Types of Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.0F3A.W0 67 /r /ib VFPCLASSSH k1{k2}, xmm1/m16, imm8	A	V/V	AVX512-FP16 OR AVX10.1[1]	Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	imm8 (r)	N/A

Description

This instruction checks the low FP16 value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against; see Table 1-15 for the categories. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in the destination mask register according to the writemask. The other bits in the destination mask register are zeroed.

Operation

VFPCLASSSH dest{k2}, src, imm8

```
IF k2[0] or *no writemask*:
    DEST.bit[0] := check_fp_class_fp16(src.fp16[0], imm8)    // see VFPCLASSPH
ELSE:
    DEST.bit[0] := 0

DEST[MAXKL-1:1] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSSH __mmask8 _mm_fpclass_sh_mask (__m128h a, int imm8);
VFPCLASSSH __mmask8 _mm_mask_fpclass_sh_mask (__mmask8 k1, __m128h a, int imm8);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Table 2-58, “Type E10 Class Exception Conditions.”

VFPCLASSSS—Tests Type of a Scalar Float32 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 67 /r VFPCLASSSS k2 {k1}, xmm2/m32, imm8	A	V/V	AVX512DQ OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

The FPCLASSSS instruction checks the low single-precision floating-point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX_KL-1: 1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 1-40. The classification test for each category is listed in Table 1-11.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
CheckFPClassSP (tsrc[31:0], imm8[7:0]){

    //* Start checking the source operand for special type *//
    NegNum  := tsrc[31];
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes := 1; FI;
    IF (tsrc[30:23]=0h) Then ExpAllZeros := 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros := 1;
    ELSIF (tsrc[22:0]=0h) Then
        MantAllZeros := 1;
    FI;
    ZeroNumber= ExpAllZeros AND MantAllZeros
    SignalingBit= tsrc[22];

    sNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res := ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite
```

```
    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
                ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
                ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
                ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} //* end of CheckSPClassSP() *//
```

VFPCLASSSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[0] :=
                CheckFPClassSP(SRC1[31:0], imm8[7:0])
    ELSE  DEST[0] := 0                    ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSSS __mmask8 _mm_fpclass_ss_mask(__m128 a, int c)
VFPCLASSSS __mmask8 _mm_mask_fpclass_ss_mask(__mmask8 m, __m128 a, int c)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-53, “Type E6 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed Dword Indices

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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Using signed dword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

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	N/A	N/A

Description

A set of single-precision/double precision faulting-point memory locations pointed by base address BASE_ADDR and index vector V_INDEX with scale SCALE are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the right most one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

Operation

```
BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement
```

VGATHERDPS (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j]
        THEN DEST[i+31:i] :=
            MEM[BASE_ADDR +
                    SignExtend(VINDEX[i+31:i]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+31:i] := remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

VGATHERDPD (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j]
        THEN DEST[i+63:i] := MEM[BASE_ADDR +
                    SignExtend(VINDEX[k+31:k]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+63:i] := remains unchanged*
```

```
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPD __m512d _mm512_i32gather_pd(__m256i vdx, void * base, int scale);
VGATHERDPD __m512d _mm512_mask_i32gather_pd(__m512d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERDPD __m256d _mm256_mmask_i32gather_pd(__m256d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPD __m128d _mm_mmask_i32gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPS __m512 _mm512_i32gather_ps(__m512i vdx, void * base, int scale);
VGATHERDPS __m512 _mm512_mask_i32gather_ps(__m512 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERDPS __m256 _mm256_mmask_i32gather_ps(__m256 s, __mmask8 k, __m256i vdx, void * base, int scale);
GATHERDPS __m128 _mm_mmask_i32gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-61, “Type E12 Class Exception Conditions.”

VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed Qword Indices

Opcode/ Instruction	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Using signed qword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

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	N/A	N/A

Description

A set of 8 single-precision/double precision faulting-point memory locations pointed by base address BASE_ADDR and index vector V_INDEX with scale SCALE are gathered. The result is written into vector a register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.

- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

Operation

```
BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement
```

VGATHERQPS (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1

    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+31:i] := remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL/2] := 0
```

VGATHERQPD (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+63:i] := remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERQPD __m512d _mm512_i64gather_pd(__m512i vdx, void * base, int scale);
VGATHERQPD __m512d _mm512_mask_i64gather_pd(__m512d s, __mmask8 k, __m512i vdx, void * base, int scale);
VGATHERQPD __m256d _mm256_mask_i64gather_pd(__m256d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPD __m128d _mm_mask_i64gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERQPS __m256 _mm512_i64gather_ps(__m512i vdx, void * base, int scale);
VGATHERQPS __m256 _mm512_mask_i64gather_ps(__m256 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERQPS __m128 _mm256_mask_i64gather_ps(__m128 s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPS __m128 _mm_mask_i64gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-61, “Type E12 Class Exception Conditions.”

VGETEXPPD—Convert Exponents of Packed Double Precision Floating-Point Values to Double Precision Floating-Point Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	Convert the exponent of packed double precision floating-point values in the source operand to double precision floating-point results representing unbiased integer exponents and stores the results in the destination register.
EVEX.256.66.0F38.W1 42 /r VGETEXPPD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert the exponent of packed double precision floating-point values in the source operand to double precision floating-point results representing unbiased integer exponents and stores the results in the destination register.
EVEX.512.66.0F38.W1 42 /r VGETEXPPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}	A	V/V	AVX512F OR AVX10.1[1]	Convert the exponent of packed double precision floating-point values in the source operand to double precision floating-point results representing unbiased integer exponents and stores the results in the destination under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Extracts the biased exponents from the normalized double precision floating-point representation of each qword data element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to double precision floating-point value and written to the corresponding qword elements of the destination operand (the first operand) as double precision floating-point numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 1-12.

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 1-12. VGETEXPPD/SD Special Cases

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	If (SRC = SNaN) then #IE If (SRC = denormal) then #DE
0 < |src1| < INF	floor($\log_2$(|src1|))	
| src1| = +INF	+INF	
| src1| = 0	-INF	

Operation

```
NormalizeExpTinyDPFP(SRC[63:0])
{
    // Jbit is the hidden integral bit of a floating-point number. In case of denormal number it has the value of ZERO.
    Src.Jbit := 0;
    Dst.exp := 1;
    Dst.fraction := SRC[51:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit := Dst.fraction[51];                // Get the fraction MSB
        Dst.fraction := Dst.fraction << 1 ;          // One bit shift left
        Dst.exp-- ;                   // Decrement the exponent
    }
    Dst.fraction := 0;                // zero out fraction bits
    Dst.sign := 1;                    // Return negative sign
    TMP[63:0] := MXCSR.DAZ? 0 : (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction) ;
    Return (TMP[63:0]);
}

ConvertExpDPFP(SRC[63:0])
{
    Src.sign := 0;                    // Zero out sign bit
    Src.exp := SRC[62:52];
    Src.fraction := SRC[51:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF ( SRC = SNAN ) SET IE;
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (Src = +INF) RETURN (Src);

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
    }
    ELSE            // check if denormal operand (notice that MXCSR.DAZ = 0)
    {
        IF ((Src.exp = 0) AND (Src.fraction != 0))
        {
            TMP[63:0] := NormalizeExpTinyDPFP(SRC[63:0]) ;        // Get Normalized Exponent
            Set #DE
        }
        ELSE            // exponent value is correct
```

```
        {
            TMP[63:0] := (Src.sign << 63) OR (Src.exp << 52) OR (Src.fraction) ;
        }
        TMP := SAR(TMP, 52) ;           // Shift Arithmetic Right
        TMP := TMP – 1023;              // Subtract Bias
        Return CvtI2D(TMP);             // Convert INT to double precision floating-point number
    }
}
```

VGETEXPPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    DEST[i+63:i] :=
            ConvertExpDPFP(SRC[63:0])
                ELSE
                    DEST[i+63:i] :=
            ConvertExpDPFP(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPPD __m512d _mm512_getexp_pd(__m512d a);
VGETEXPPD __m512d _mm512_mask_getexp_pd(__m512d s, __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_maskz_getexp_pd(__mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_getexp_round_pd(__m512d a, int sae);
VGETEXPPD __m512d _mm512_mask_getexp_round_pd(__m512d s, __mmask8 k, __m512d a, int sae);
VGETEXPPD __m512d _mm512_maskz_getexp_round_pd(__mmask8 k, __m512d a, int sae);
VGETEXPPD __m256d _mm256_getexp_pd(__m256d a);
VGETEXPPD __m256d _mm256_mask_getexp_pd(__m256d s, __mmask8 k, __m256d a);
VGETEXPPD __m256d _mm256_maskz_getexp_pd(__mmask8 k, __m256d a);
VGETEXPPD __m128d _mm_getexp_pd(__m128d a);
VGETEXPPD __m128d _mm_mask_getexp_pd(__m128d s, __mmask8 k, __m128d a);
VGETEXPPD __m128d _mm_maskz_getexp_pd(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VGETEXPPH—Convert Exponents of Packed FP16 Values to FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 42 /r VGETEXPPH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert the exponent of FP16 values in the source operand to FP16 results representing unbiased integer exponents and stores the results in the destination register subject to writemask k1.
EVEX.256.66.MAP6.W0 42 /r VGETEXPPH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Convert the exponent of FP16 values in the source operand to FP16 results representing unbiased integer exponents and stores the results in the destination register subject to writemask k1.
EVEX.512.66.MAP6.W0 42 /r VGETEXPPH zmm1{k1}{z}, zmm2/m512/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert the exponent of FP16 values in the source operand to FP16 results representing unbiased integer exponents and stores the results in the destination register subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction extracts the biased exponents from the normalized FP16 representation of each word element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to an FP16 value and written to the corresponding word elements of the destination operand (the first operand) as FP16 numbers.

The destination elements are updated according to the writemask.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 1-11.

The formula is:

GETEXP(x) = floor($\log_2(|x|)$)

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPH). Thus, the VGETEXPPH instruction does not require software to handle SIMD floating-point exceptions.

Table 1-13. VGETEXPPH/VGETEXPSH Special Cases

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	If (SRC = SNaN), then #IE. If (SRC = denormal), then #DE.
0 < |src1| < INF	floor($\log_2$(|src1|))	
| src1| = +INF	+INF	
| src1| = 0	-INF	

Operation

```
def normalize_exponent_tiny_fp16(src):
    jbit := 0
    // src & dst are FP16 numbers with sign(1b), exp(5b) and fraction (10b) fields
    dst.exp := 1                    // write bits 14:10
    dst.fraction := src.fraction // copy bits 9:0
    while jbit == 0:
        jbit := dst.fraction[9]     // msb of the fraction
        dst.fraction := dst.fraction << 1
        dst.exp := dst.exp - 1
    dst.fraction := 0
    return dst

def getexp_fp16(src):
    src.sign := 0                   // make positive
    exponent_all_ones := (src[14:10] == 0x1F)
    exponent_all_zeros := (src[14:10] == 0)
    mantissa_all_zeros := (src[9:0] == 0)
    zero := exponent_all_zeros and mantissa_all_zeros
    signaling_bit := src[9]

    nan := exponent_all_ones and not(mantissa_all_zeros)
    snan := nan and not(signaling_bit)
    qnan := nan and signaling_bit
    positive_infinity := not(negative) and exponent_all_ones and mantissa_all_zeros
    denormal := exponent_all_zeros and not(mantissa_all_zeros)

    if nan:
        if snan:
            MXCSR.IE := 1
        return qnan(src)            // convert snan to a qnan
    if positive_infinity:
        return src
    if zero:
        return -INF
    if denormal:
        tmp := normalize_exponent_tiny_fp16(src)
        MXCSR.DE := 1
    else:
        tmp := src
    tmp := SAR(tmp, 10)             // shift arithmetic right
    tmp := tmp - 15                 // subtract bias
    return convert_integer_to_fp16(tmp)
```

```
VGETEXPPH dest{k1}, src
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := getexp_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPPH __m128h _mm_getexp_ph (__m128h a);
VGETEXPPH __m128h _mm_mask_getexp_ph (__m128h src, __mmask8 k, __m128h a);
VGETEXPPH __m128h _mm_maskz_getexp_ph (__mmask8 k, __m128h a);
VGETEXPPH __m256h _mm256_getexp_ph (__m256h a);
VGETEXPPH __m256h _mm256_mask_getexp_ph (__m256h src, __mmask16 k, __m256h a);
VGETEXPPH __m256h _mm256_maskz_getexp_ph (__mmask16 k, __m256h a);
VGETEXPPH __m512h _mm512_getexp_ph (__m512h a);
VGETEXPPH __m512h _mm512_mask_getexp_ph (__m512h src, __mmask32 k, __m512h a);
VGETEXPPH __m512h _mm512_maskz_getexp_ph (__mmask32 k, __m512h a);
VGETEXPPH __m512h _mm512_getexp_round_ph (__m512h a, const int sae);
VGETEXPPH __m512h _mm512_mask_getexp_round_ph (__m512h src, __mmask32 k, __m512h a, const int sae);
VGETEXPPH __m512h _mm512_maskz_getexp_round_ph (__mmask32 k, __m512h a, const int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VGETEXPPS—Convert Exponents of Packed Single Precision Floating-Point Values to Single Precision Floating-Point Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	Convert the exponent of packed single-precision floating-point values in the source operand to single-precision floating-point results representing unbiased integer exponents and stores the results in the destination register.
EVEX.256.66.0F38.W0 42 /r VGETEXPPS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Convert the exponent of packed single-precision floating-point values in the source operand to single-precision floating-point results representing unbiased integer exponents and stores the results in the destination register.
EVEX.512.66.0F38.W0 42 /r VGETEXPPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}	A	V/V	AVX512F OR AVX10.1[1]	Convert the exponent of packed single-precision floating-point values in the source operand to single-precision floating-point results representing unbiased integer exponents and stores the results in the destination register.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Extracts the biased exponents from the normalized single-precision floating-point representation of each dword element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to single-precision floating-point value and written to the corresponding dword elements of the destination operand (the first operand) as single-precision floating-point numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 1-14.

The formula is:

GETEXP(x) = floor($\log_2$(|x|))

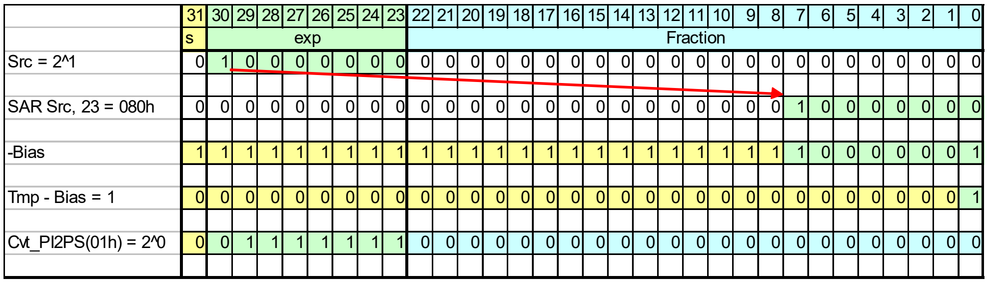
Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD floating-point exceptions.

Table 1-14. 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$_2$(|src1|))	
| src1| = +INF	+INF	
| src1| = 0	-INF	

Figure 1-41 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 = 2^1	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
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	1	0	0	0	0	0	0	0
-Bias	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1
Tmp - Bias = 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Cvt_PI2PS(01h) = 2^0	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

Figure 1-41. VGETEXPPS Functionality on Normal Input Values

Operation

```
NormalizeExpTinySPFP(SRC[31:0])
{
    // Jbit is the hidden integral bit of a floating-point number. In case of denormal number it has the value of ZERO.
    Src.Jbit := 0;
    Dst.exp := 1;
    Dst.fraction := SRC[22:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit := Dst.fraction[22];           // Get the fraction MSB
        Dst.fraction := Dst.fraction << 1 ;     // One bit shift left
        Dst.exp-- ;                 // Decrement the exponent
    }
    Dst.fraction := 0;              // zero out fraction bits
    Dst.sign := 1;                  // Return negative sign
    TMP[31:0] := MXCSR.DAZ? 0 : (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction) ;
    Return (TMP[31:0]);
}
ConvertExpSPFP(SRC[31:0])
{
    Src.sign := 0;                  // Zero out sign bit
    Src.exp := SRC[30:23];
    Src.fraction := SRC[22:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF ( SRC = SNAN ) SET IE;
```

```
            Return QNAN(SRC);
    }
    // Check for +INF
    IF (Src = +INF) RETURN (Src);

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
    }
    ELSE            // check if denormal operand (notice that MXCSR.DAZ = 0)
    {
        IF ((Src.exp = 0) AND (Src.fraction != 0))
        {
            TMP[31:0] := NormalizeExpTinySPFP(SRC[31:0]) ;      // Get Normalized Exponent
            Set #DE
        }
        ELSE            // exponent value is correct
        {
            TMP[31:0] := (Src.sign << 31) OR (Src.exp << 23) OR (Src.fraction) ;
        }
        TMP := SAR(TMP, 23) ;           // Shift Arithmetic Right
        TMP := TMP – 127;               // Subtract Bias
        Return CvtI2S(TMP);             // Convert INT to single precision floating-point number
    }
}
```

VGETEXPPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    DEST[i+31:i] :=
            ConvertExpSPFP(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            ConvertExpSPFP(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPPS __m512 _mm512_getexp_ps(__m512 a);
VGETEXPPS __m512 _mm512_mask_getexp_ps(__m512 s, __mmask16 k, __m512 a);
VGETEXPPS __m512 _mm512_maskz_getexp_ps(__mmask16 k, __m512 a);
VGETEXPPS __m512 _mm512_getexp_round_ps(__m512 a, int sae);
VGETEXPPS __m512 _mm512_mask_getexp_round_ps(__m512 s, __mmask16 k, __m512 a, int sae);
VGETEXPPS __m512 _mm512_maskz_getexp_round_ps(__mmask16 k, __m512 a, int sae);
VGETEXPPS __m256 _mm256_getexp_ps(__m256 a);
VGETEXPPS __m256 _mm256_mask_getexp_ps(__m256 s, __mmask8 k, __m256 a);
VGETEXPPS __m256 _mm256_maskz_getexp_ps(__mmask8 k, __m256 a);
VGETEXPPS __m128 _mm_getexp_ps(__m128 a);
VGETEXPPS __m128 _mm_mask_getexp_ps(__m128 s, __mmask8 k, __m128 a);
VGETEXPPS __m128 _mm_maskz_getexp_ps(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-46, "Type E2 Class Exception Conditions."

Additionally:

#UD If EVEX.vvvv != 1111B.

VGETEXPSD—Convert Exponents of Scalar Double Precision Floating-Point Value to Double Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W1 43 /r VGETEXPSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	A	V/V	AVX512F OR AVX10.1[1]	Convert the biased exponent (bits 62:52) of the low double precision floating-point value in xmm3/m64 to a double precision floating-point value representing unbiased integer exponent. Stores the result to the low 64-bit of xmm1 under the writemask k1 and merge with the other elements of xmm2.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Extracts the biased exponent from the normalized double precision floating-point representation of the low qword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to double precision floating-point value and written to the destination operand (the first operand) as double precision floating-point numbers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float64 memory location.

If writemasking is used, the low quadword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low quadword element of the destination operand is unconditionally updated.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 1-12.

The formula is:

$\mathrm{GETEXP}(x) = \mathrm{floor}(\log_2(|x|))$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Operation

```
// NormalizeExpTinyDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

// ConvertExpDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD
```

VGETEXPSD (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[63:0] :=
            ConvertExpDPFP(SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
```

```
            DEST[63:0] := 0
    FI
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPSD __m128d _mm_getexp_sd(__m128d a, __m128d b);
VGETEXPSD __m128d _mm_mask_getexp_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d _mm_maskz_getexp_sd(__mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d _mm_getexp_round_sd(__m128d a, __m128d b, int sae);
VGETEXPSD __m128d _mm_mask_getexp_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int sae);
VGETEXPSD __m128d _mm_maskz_getexp_round_sd(__mmask8 k, __m128d a, __m128d b, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Table 2-47, "Type E3 Class Exception Conditions."

VGETEXPSH—Convert Exponents of Scalar FP16 Values to FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 43 /r VGETEXPSH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Convert the exponent of FP16 values in the low word of the source operand to FP16 results representing unbiased integer exponents, and stores the results in the low word of the destination register subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction extracts the biased exponents from the normalized FP16 representation of the low word element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to an unbiased negative integer value. The integer value of the unbiased exponent is converted to an FP16 value and written to the low word element of the destination operand (the first operand) as an FP16 number.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 1-13.

The formula is:

$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTSH). Thus, the VGETEXPSH instruction does not require software to handle SIMD floating-point exceptions.

Operation

VGETEXPSH dest{k1}, src1, src2

```
IF k1[0] or *no writemask*:
    DEST.fp16[0] := getexp_fp16(src2.fp16[0]) // see VGETEXPPH
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPSH __m128h _mm_getexp_round_sh (__m128h a, __m128h b, const int sae);
VGETEXPSH __m128h _mm_mask_getexp_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, const int sae);
VGETEXPSH __m128h _mm_maskz_getexp_round_sh (__mmask8 k, __m128h a, __m128h b, const int sae);
VGETEXPSH __m128h _mm_getexp_sh (__m128h a, __m128h b);
VGETEXPSH __m128h _mm_mask_getexp_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VGETEXPSH __m128h _mm_maskz_getexp_sh (__mmask8 k, __m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VGETEXPSS—Convert Exponents of Scalar Single Precision Floating-Point Value to Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 43 /r VGETEXPSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	A	V/V	AVX512F OR AVX10.1[1]	Convert the biased exponent (bits 30:23) of the low single-precision floating-point value in xmm3/m32 to a single-precision floating-point value representing unbiased integer exponent. Stores the result to xmm1 under the writemask k1 and merge with the other elements of xmm2.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Extracts the biased exponent from the normalized single-precision floating-point representation of the low doubleword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to single-precision floating-point value and written to the destination operand (the first operand) as single-precision floating-point numbers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float32 memory location.

If writemasking is used, the low doubleword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low doubleword element of the destination operand is unconditionally updated.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 1-14.

The formula is:

$\mathrm{GETEXP}(x) = \mathrm{floor}(\log_2(|x|))$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD floating-point exceptions.

Operation

```
// NormalizeExpTinySPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS
// ConvertExpSPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS
```

VGETEXPSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[31:0] :=
            ConvertExpDPFP(SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
```

```
                THEN *DEST[31:0] remains unchanged*
                ELSE                                    ; zeroing-masking
                      DEST[31:0]:= 0
                FI
    FI;
ENDFOR
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETEXPSS __m128 _mm_getexp_ss( __m128 a, __m128 b);
VGETEXPSS __m128 _mm_mask_getexp_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_maskz_getexp_ss( __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_getexp_round_ss( __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_mask_getexp_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_maskz_getexp_round_ss( __mmask8 k, __m128 a, __m128 b, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions.”

VGETMANTPD—Extract Float64 Vector of Normalized Mantissas From Float64 Vector

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	Get Normalized Mantissa from float64 vector xmm2/m128/m64bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.256.66.0F3A.W1 26 /r ib VGETMANTPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Get Normalized Mantissa from float64 vector ymm2/m256/m64bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.512.66.0F3A.W1 26 /r ib VGETMANTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	A	V/V	AVX512F OR AVX10.1[1]	Get Normalized Mantissa from float64 vector zmm2/m512/m64bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Convert double precision floating values in the source operand (the second operand) to double precision floating-point values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 1-42. 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.

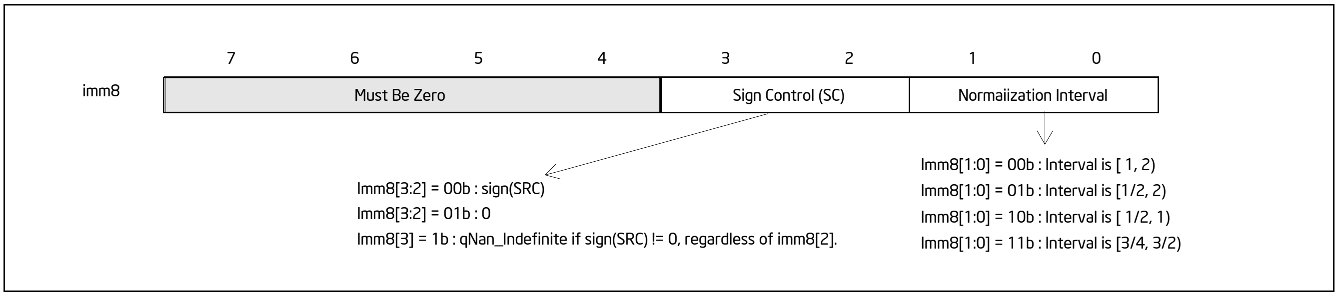


Figure 1-42. Imm8 Controls for VGETMANTPD/SD/PS/SS

For each input double precision floating-point value x, The conversion operation is:

$$\mathrm{GetMant}(x) = \pm 2^k|x.\mathrm{significand}|$$

where:

1 <= |x.significand| < 2

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 1-42.

Each converted double precision floating-point result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 1-15 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 1-15. GetMant() Special Float Values Behavior

Input	Result	Exceptions / Comments
NaN	QNaN(SRC)	Ignore *interv* If (SRC = SNaN) then #IE
+?	1.0	Ignore *interv*
+0	1.0	Ignore *interv*
-0	IF (SC[0]) THEN +1.0 ELSE -1.0	Ignore *interv*

-?	IF (SC[1]) THEN {QNaN_Indefinite}
	ELSE {
	IF (SC[0]) THEN +1.0
	ELSE -1.0
		Ignore *interv*
		If (SC[1]) then #IE

negative	SC[1] ? QNaN_Indefinite : Getmant(SRC)[1]	If (SC[1]) then #IE

NOTES:

1. In case SC[1]==0, the sign of Getmant(SRC) is declared according to SC[0].

Operation

```
def getmant_fp64(src, sign_control, normalization_interval):
    bias := 1023
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and ((dst.fraction = 0) or (MXCSR.DAZ=1))
    denormal := (dst.exp = 0) and (dst.fraction != 0) and (MXCSR.DAZ=0)
    infinity := (dst.exp = 0x7FF) and (dst.fraction = 0)
    nan := (dst.exp = 0x7FF) and (dst.fraction != 0)
    src_signaling := src.fraction[51]
    snan := nan and (src_signaling = 0)
    positive := (src.sign = 0)
    negative := (src.sign = 1)
    if nan:
        if snan:
            MXCSR.IE := 1
        return qnan(src)
```

```
    if positive and (zero or infinity):
        return 1.0
    if negative:
        if zero:
            return signed_one
        if infinity:
            if sign_control[1]:
                MXCSR.IE := 1
                return QNaN_Indefinite
            return signed_one
        if sign_control[1]:
            MXCSR.IE := 1
            return QNaN_Indefinite

    if denormal:
        jbit := 0
        dst.exp := bias
        while jbit = 0:
            jbit := dst.fraction[51]
            dst.fraction := dst.fraction << 1
            dst.exp : = dst.exp - 1
        MXCSR.DE := 1

    unbiased_exp := dst.exp - bias
    odd_exp := unbiased_exp[0]
    signaling_bit := dst.fraction[51]
    if normalization_interval = 0b00:
        dst.exp := bias
    else if normalization_interval = 0b01:
        dst.exp := odd_exp ? bias-1 : bias
    else if normalization_interval = 0b10:
        dst.exp := bias-1
    else if normalization_interval = 0b11:
        dst.exp := signaling_bit ? bias-1 : bias
    return dst
```

VGETMANTPD (EVEX Encoded Versions)

```
VGETMANTPD dest{k1}, src, imm8
VL = 128, 256, or 512
KL := VL / 64
sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.double[0]
        ELSE:
            tsrc := src.double[i]
        DEST.double[i] := getmant_fp64(tsrc, sign_control, normalization_interval)
    ELSE IF *zeroing*:
        DEST.double[i] := 0
    //else DEST.double[i] remains unchanged
```

```
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETMANTPD __m512d _mm512_getmant_pd( __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_mask_getmant_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_maskz_getmant_pd( __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_getmant_round_pd( __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_mask_getmant_round_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_maskz_getmant_round_pd( __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m256d _mm256_getmant_pd( __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_mask_getmant_pd(__m256d s, __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_maskz_getmant_pd( __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_getmant_pd( __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_mask_getmant_pd(__m128d s, __mmask8 k, __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_maskz_getmant_pd( __mmask8 k, __m128d a, enum intv, enum sgn);
```

SIMD Floating-Point Exceptions

Denormal, Invalid.

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VGETMANTPH—Extract FP16 Vector of Normalized Mantissas from FP16 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.0F3A.W0 26 /r /ib VGETMANTPH xmm1{k1}{z}, xmm2/m128/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Get normalized mantissa from FP16 vector xmm2/m128/m16bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, subject to writemask k1.
EVEX.256.NP.0F3A.W0 26 /r /ib VGETMANTPH ymm1{k1}{z}, ymm2/m256/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Get normalized mantissa from FP16 vector ymm2/m256/m16bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, subject to writemask k1.
EVEX.512.NP.0F3A.W0 26 /r /ib VGETMANTPH zmm1{k1}{z}, zmm2/m512/m16bcst {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1[1]	Get normalized mantissa from FP16 vector zmm2/m512/m16bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8 (r)	N/A

Description

This instruction converts the FP16 values in the source operand (the second operand) to FP16 values with the mantissa normalization and sign control specified by the imm8 byte, see Table 1-16. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (SC) is specified by bits 3:2 of the immediate byte.

The destination elements are updated according to the writemask.

Table 1-16. imm8 Controls for VGETMANTPH/VGETMANTSH

imm8 Bits	Definition
imm8[7:4]	Must be zero.
imm8[3:2]	Sign Control (SC) 0b00: Sign(SRC) 0b01: 0 0b1x: QNaN_Indefinite if sign(SRC)!=0
imm8[1:0]	Interv 0b00: Interval is [1, 2) 0b01: Interval is [1/2, 2) 0b10: Interval is [1/2, 1) 0b11: Interval is [3/4, 3/2)

For each input FP16 value x, The conversion operation is:

$$GetMant(x) = \pm 2^k |x.significand|$$

where:

$1 \leq |x.\text{significand}| < 2$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by the sign control and the source sign and the leading fraction bit.

The encoded value of imm8[1:0] and sign control are shown in Table 1-16.

Each converted FP16 result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 1-17 when dealing with floating-point special numbers.

Table 1-17. GetMant() Special Float Values Behavior

Input	Result	Exceptions / Comments
NaN	QNaN(SRC)	Ignore *interv.* If (SRC = SNaN), then #IE.
+∞	1.0	Ignore *interv.*
+0	1.0	Ignore *interv.*
-0	IF (SC[0]) THEN +1.0 ELSE -1.0	Ignore *interv.*

-∞	IF (SC[1]) THEN {QNaN_Indefinite}
	ELSE {
	IF (SC[0]) THEN +1.0
	ELSE -1.0
		Ignore *interv.*
		If (SC[1]), then #IE.

negative	SC[1] ? QNaN_Indefinite : Getmant(SRC)[1]	If (SC[1]), then #IE.

NOTES:

1. In case SC[1]==0, the sign of Getmant(SRC) is declared according to SC[0].

Operation

```
def getmant_fp16(src, sign_control, normalization_interval):
    bias := 15
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and (dst.fraction = 0)
    denormal := (dst.exp = 0) and (dst.fraction != 0)
    infinity := (dst.exp = 0x1F) and (dst.fraction = 0)
    nan := (dst.exp = 0x1F) and (dst.fraction != 0)
    src_signaling := src.fraction[9]
    snan := nan and (src_signaling = 0)
    positive := (src.sign = 0)
    negative := (src.sign = 1)
    if nan:
        if snan:
            MXCSR.IE := 1
        return qnan(src)

    if positive and (zero or infinity):
        return 1.0
    if negative:
        if zero:
```

```
                return signed_one
            if infinity:
                if sign_control[1]:
                    MXCSR.IE := 1
                    return QNaN_Indefinite
                return signed_one
            if sign_control[1]:
                MXCSR.IE := 1
                return QNaN_Indefinite
        if denormal:
            jbit := 0
            dst.exp := bias             // set exponent to bias value
            while jbit = 0:
                jbit := dst.fraction[9]
                dst.fraction := dst.fraction << 1
                dst.exp : = dst.exp - 1
            MXCSR.DE := 1

        unbaiased_exp := dst.exp - bias
        odd_exp := unbaiased_exp[0]
        signaling_bit := dst.fraction[9]
        if normalization_interval = 0b00:
            dst.exp := bias
        else if normalization_interval = 0b01:
            dst.exp := odd_exp ? bias-1 : bias
        else if normalization_interval = 0b10:
            dst.exp := bias-1
        else if normalization_interval = 0b11:
            dst.exp := signaling_bit ? bias-1 : bias
        return dst
```

VGETMANTPH dest{k1}, src, imm8

```
VL = 128, 256 or 512
KL := VL/16

sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := getmant_fp16(tsrc, sign_control, normalization_interval)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTPH __m128h _mm_getmant_ph (__m128h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m128h _mm_mask_getmant_ph (__m128h src, __mmask8 k, __m128h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m128h _mm_maskz_getmant_ph (__mmask8 k, __m128h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m256h _mm256_getmant_ph (__m256h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m256h _mm256_mask_getmant_ph (__m256h src, __mmask16 k, __m256h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m256h _mm256_maskz_getmant_ph (__mmask16 k, __m256h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_getmant_ph (__m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_mask_getmant_ph (__m512h src, __mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_maskz_getmant_ph (__mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_getmant_round_ph (__m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);
VGETMANTPH __m512h _mm512_mask_getmant_round_ph (__m512h src, __mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);
VGETMANTPH __m512h _mm512_maskz_getmant_round_ph (__mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."

VGETMANTPS—Extract Float32 Vector of Normalized Mantissas From Float32 Vector

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Convert single-precision floating values in the source operand (the second operand) to single-precision floating-point values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 1-42. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

For each input single-precision floating-point value x, The conversion operation is:

$$GetMant(x) = \pm 2^k |x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 1-42.

Each converted single-precision floating-point result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 1-15 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Note: EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

```
def getmant_fp32(src, sign_control, normalization_interval):
    bias := 127
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and ((dst.fraction = 0) or (MXCSR.DAZ=1))
    denormal := (dst.exp = 0) and (dst.fraction != 0) and (MXCSR.DAZ=0)
    infinity := (dst.exp = 0xFF) and (dst.fraction = 0)
    nan := (dst.exp = 0xFF) and (dst.fraction != 0)
    src_signaling := src.fraction[22]
    snan := nan and (src_signaling = 0)
    positive := (src.sign = 0)
    negative := (src.sign = 1)
    if nan:
        if snan:
            MXCSR.IE := 1
        return qnan(src)

    if positive and (zero or infinity):
        return 1.0
    if negative:
        if zero:
            return signed_one
        if infinity:
            if sign_control[1]:
                MXCSR.IE := 1
                return QNaN_Indefinite
            return signed_one
        if sign_control[1]:
            MXCSR.IE := 1
            return QNaN_Indefinite

    if denormal:
        jbit := 0
        dst.exp := bias
        while jbit = 0:
            jbit := dst.fraction[22]
            dst.fraction := dst.fraction << 1
            dst.exp : = dst.exp - 1
        MXCSR.DE := 1

    unbiased_exp := dst.exp - bias
    odd_exp  := unbiased_exp[0]
    signaling_bit := dst.fraction[22]
    if normalization_interval = 0b00:
        dst.exp := bias
    else if normalization_interval = 0b01:
        dst.exp := odd_exp ? bias-1 : bias
    else if normalization_interval = 0b10:
        dst.exp := bias-1
    else if normalization_interval = 0b11:
        dst.exp := signaling_bit ? bias-1 : bias
```

```
    return dst
```

VGETMANTPS (EVEX Encoded Versions)

```
VGETMANTPS dest{k1}, src, imm8
VL = 128, 256, or 512
KL := VL / 32
sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.float[0]
        ELSE:
            tsrc := src.float[i]
        DEST.float[i] := getmant_fp32(tsrc, sign_control, normalization_interval)
    ELSE IF *zeroing*:
        DEST.float[i] := 0
    //else DEST.float[i] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTPS __m512 _mm512_getmant_ps(__m512 a, enum intv, enum sgn);
VGETMANTPS __m512 _mm512_mask_getmant_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn;
VGETMANTPS __m512 _mm512_maskz_getmant_ps(__mmask16 k, __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 _mm512_getmant_round_ps(__m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m512 _mm512_mask_getmant_round_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m512 _mm512_maskz_getmant_round_ps(__mmask16 k, __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m256 _mm256_getmant_ps(__m256 a, enum intv, enum sgn);
VGETMANTPS __m256 _mm256_mask_getmant_ps(__m256 s, __mmask8 k, __m256 a, enum intv, enum sgn);
VGETMANTPS __m256 _mm256_maskz_getmant_ps(__mmask8 k, __m256 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_getmant_ps(__m128 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_mask_getmant_ps(__m128 s, __mmask8 k, __m128 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_maskz_getmant_ps(__mmask8 k, __m128 a, enum intv, enum sgn);

SIMD Floating-Point Exceptions

Denormal, Invalid.

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VGETMANTSD—Extract Float64 of Normalized Mantissas From Float64 Scalar

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 27 /r ib VGETMANTSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F OR AVX10.1[1]	Extract the normalized mantissa of the low float64 element in xmm3/m64 using imm8 for sign control and mantissa interval normalization. Store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Convert the double precision floating values in the low quadword element of the second source operand (the third operand) to double precision floating-point value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 1-42. The converted result is written to the low quadword element of the destination operand (the first operand) using writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

$$GetMant(x) = \pm 2^k |x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 1-42.

The converted double precision floating-point result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 1-15 when dealing with floating-point special numbers.

If writemasking is used, the low quadword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low quadword element of the destination operand is unconditionally updated.

Operation

```
// getmant_fp64(src, sign_control, normalization_interval) is defined in the operation section of VGETMANTPD
```

VGETMANTSD (EVEX encoded version)

```
SignCtrl[1:0] := IMM8[3:2];
Interv[1:0] := IMM8[1:0];
IF k1[0] OR *no writemask*
    THEN DEST[63:0] :=
            getmant_fp64(src, sign_control, normalization_interval)
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] := 0
        FI
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETMANTSD __m128d _mm_getmant_sd( __m128d a, __m128 b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_mask_getmant_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_maskz_getmant_sd( __mmask8 k, __m128 a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_getmant_round_sd( __m128d a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_mask_getmant_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_maskz_getmant_round_sd( __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
```

SIMD Floating-Point Exceptions

Denormal, Invalid.

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions.”

VGETMANTSH—Extract FP16 of Normalized Mantissa from FP16 Scalar

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.0F3A.W0 27 /r /ib VGETMANTSH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1[1]	Extract the normalized mantissa of the low FP16 element in xmm3/m16 using imm8 for sign control and mantissa interval normalization. Store the mantissa to xmm1 subject to writemask k1 and merge with the other elements of xmm2. Bits 127:16 of xmm2 are copied to xmm1[127:16].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

This instruction converts the FP16 value in the low element of the second source operand to FP16 values with the mantissa normalization and sign control specified by the imm8 byte, see Table 1-16. The converted result is written to the low element of the destination operand using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (SC) is specified by bits 3:2 of the immediate byte.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

For each input FP16 value x, The conversion operation is:

$$GetMant(x) = \pm 2^k |x.significand|$$

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 the sign control and the source sign and the leading fraction bit.

The encoded value of imm8[1:0] and sign control are shown in Table 1-16.

Each converted FP16 result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 1-17 when dealing with floating-point special numbers.

Operation

VGETMANTSH dest{k1}, src1, src2, imm8

```
sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

IF k1[0] or *no writemask*:
    dest.fp16[0] := getmant_fp16(src2.fp16[0],      // see VGETMANTPH
                                 sign_control,
                                 normalization_interval)
ELSE IF *zeroing*:
    dest.fp16[0] := 0
```

```
//else dest.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTSH __m128h _mm_getmant_round_sh (__m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);

VGETMANTSH __m128h _mm_mask_getmant_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);

VGETMANTSH __m128h _mm_maskz_getmant_round_sh (__mmask8 k, __m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);

VGETMANTSH __m128h _mm_getmant_sh (__m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);

VGETMANTSH __m128h _mm_mask_getmant_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);

VGETMANTSH __m128h _mm_maskz_getmant_sh (__mmask8 k, __m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VGETMANTSS—Extract Float32 Vector of Normalized Mantissa From Float32 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 27 /r ib VGETMANTSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Convert the single-precision floating values in the low doubleword element of the second source operand (the third operand) to single-precision floating-point value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 1-42. 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:

$$1 <= |x.\text{significand}| < 2$$

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 1-42.

The converted single-precision floating-point result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 1-15 when dealing with floating-point special numbers.

If writemasking is used, the low doubleword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low doubleword element of the destination operand is unconditionally updated.

Operation

// getmant_fp32(src, sign_control, normalization_interval) is defined in the operation section of VGETMANTPS

VGETMANTSS (EVEX encoded version)

```
SignCtrl[1:0] := IMM8[3:2];
Interv[1:0] := IMM8[1:0];
IF k1[0] OR *no writemask*
    THEN DEST[31:0] :=
            getmant_fp32(src, sign_control, normalization_interval)
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] := 0
        FI
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETMANTSS __m128 _mm_getmant_ss( __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_mask_getmant_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_maskz_getmant_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_getmant_round_ss( __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_mask_getmant_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_maskz_getmant_round_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
```

SIMD Floating-Point Exceptions

Denormal, Invalid.

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions.”

VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	Insert 256 bits of packed double precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
B	Tuple2	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple4	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8
D	Tuple8	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

VINSERTF128/VINSERTF32x4 and VINSERTF64x2 insert 128-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granularity offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination operand are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The destination and first source operands are vector registers.

VINSERTF32x4: The destination operand is a ZMM/YMM register and updated at 32-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF64x2: The destination operand is a ZMM/YMM register and updated at 64-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF32x8 and VINSERTF64x4 inserts 256-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The high 7 bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32/64-bit granularity according to the writemask.

Operation

VINSERTF32x4 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0:    TMP_DEST[127:0] := SRC2[127:0]
        1:    TMP_DEST[255:128] := SRC2[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00:  TMP_DEST[127:0] := SRC2[127:0]
        01:  TMP_DEST[255:128] := SRC2[127:0]
        10:  TMP_DEST[383:256] := SRC2[127:0]
        11:  TMP_DEST[511:384] := SRC2[127:0]
    ESAC.
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTF64x2 (EVEX encoded versions)

```
(KL, VL) = (4, 256), (8, 512)
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0:    TMP_DEST[127:0] := SRC2[127:0]
        1:    TMP_DEST[255:128] := SRC2[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
```

```
            00:  TMP_DEST[127:0] := SRC2[127:0]
            01:  TMP_DEST[255:128] := SRC2[127:0]
            10:  TMP_DEST[383:256] := SRC2[127:0]
            11:  TMP_DEST[511:384] := SRC2[127:0]
        ESAC.
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTF32x8 (EVEX.U1.512 encoded version)

```
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 15
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTF64x4 (EVEX.512 encoded version)

```
VL = 512
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 7
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
```

```
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTF128 (VEX encoded version)

```
TEMP[255:0] := SRC1[255:0]
CASE (imm8[0]) OF
    0: TEMP[127:0] := SRC2[127:0]
    1: TEMP[255:128] := SRC2[127:0]
ESAC
DEST := TEMP
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VINSERTF32x4 __m512 _mm512_insertf32x4( __m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 _mm512_mask_insertf32x4(__m512 s, __mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 _mm512_maskz_insertf32x4( __mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_insertf32x4( __m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_mask_insertf32x4(__m256 s, __mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_maskz_insertf32x4( __mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x8 __m512 _mm512_insertf32x8( __m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 _mm512_mask_insertf32x8(__m512 s, __mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 _mm512_maskz_insertf32x8( __mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF64x2 __m512d _mm512_insertf64x2( __m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d _mm512_mask_insertf64x2(__m512d s, __mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d _mm512_maskz_insertf64x2( __mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x2 __m256d _mm256_insertf64x2( __m256d a, __m128d b, int imm);
VINSERTF64x2 __m256d _mm256_mask_insertf64x2(__m256d s, __mmask8 k, __m256d a, __m128d b, int imm);
VINSERTF64x2 __m256d _mm256_maskz_insertf64x2( __mmask8 k, __m256d a, __m128d b, int imm);
VINSERTF64x4 __m512d _mm512_insertf64x4( __m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d _mm512_mask_insertf64x4(__m512d s, __mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d _mm512_maskz_insertf64x4( __mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF128 __m256 _mm256_insertf128_ps (__m256 a, __m128 b, int offset);
VINSERTF128 __m256d _mm256_insertf128_pd (__m256d a, __m128d b, int offset);
VINSERTF128 __m256i _mm256_insertf128_si256 (__m256i a, __m128i b, int offset);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-23, “Type 6 Class Exception Conditions.”

Additionally:

#UD If VEX.L = 0.

EVEX-encoded instruction, see Table 2-54, “Type E6NF Class Exception Conditions.”

VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	Insert 256 bits of packed quadword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
B	Tuple2	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple4	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8
D	Tuple8	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

VINSERTI32x4 and VINSERTI64x2 inserts 128-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granular offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 6/7bits of the immediate are ignored. The destination operand is a ZMM/YMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI32x8 and VINSERTI64x4 inserts 256-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The upper bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI128 inserts 128-bits of packed integer data from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 7 bits of the immediate are ignored. VEX.L must be 1, otherwise attempt to execute this instruction with VEX.L=0 will cause #UD.

Operation

VINSERTI32x4 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0:   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 _mm512_inserti64x4(__m512i a, __m256i b, int imm);
VINSERTI64x4 _mm512_mask_inserti64x4(__m512i s, __mmask8 k, __m512i a, __m256i b, int imm);
VINSERTI64x4 _mm512_maskz_inserti64x4(__mmask m, __m512i a, __m256i b, int imm);
VINSERTI128 __m256i _mm256_insertf128_si256 (__m256i a, __m128i b, int offset);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-23, “Type 6 Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Table 2-54, “Type E6NF Class Exception Conditions.”

VMAXPH—Return Maximum of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5F /r VMAXPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Return the maximum packed FP16 values between xmm2 and xmm3/m128/m16bcst and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5F /r VMAXPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Return the maximum packed FP16 values between ymm2 and ymm3/m256/m16bcst and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5F /r VMAXPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Return the maximum packed FP16 values between zmm2 and zmm3/m512/m16bcst and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD compare of the packed FP16 values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMAXPH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcast from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

```
def MAX(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 > SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2
```

```
VMAXPH dest, src1, src2
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            tsrc2 := SRC2.fp16[0]
        ELSE:
            tsrc2 := SRC2.fp16[j]
        DEST.fp16[j] := MAX(SRC1.fp16[j], tsrc2)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VMAXPH __m128h _mm_mask_max_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMAXPH __m128h _mm_maskz_max_ph (__mmask8 k, __m128h a, __m128h b);
VMAXPH __m128h _mm_max_ph (__m128h a, __m128h b);
VMAXPH __m256h _mm256_mask_max_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VMAXPH __m256h _mm256_maskz_max_ph (__mmask16 k, __m256h a, __m256h b);
VMAXPH __m256h _mm256_max_ph (__m256h a, __m256h b);
VMAXPH __m512h _mm512_mask_max_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VMAXPH __m512h _mm512_maskz_max_ph (__mmask32 k, __m512h a, __m512h b);
VMAXPH __m512h _mm512_max_ph (__m512h a, __m512h b);
VMAXPH __m512h _mm512_mask_max_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int sae);
VMAXPH __m512h _mm512_maskz_max_round_ph (__mmask32 k, __m512h a, __m512h b, int sae);
VMAXPH __m512h _mm512_max_round_ph (__m512h a, __m512h b, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VMAXSH—Return Maximum of Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 5F /r VMAXSH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Return the maximum low FP16 value between xmm3/m16 and xmm2 and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

NOTES:

[1] For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a compare of the low packed FP16 values in the first source operand and the second source operand and returns the maximum value for the pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMAXSH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

```
def MAX(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 > SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2
```

VMAXSH dest, src1, src2

```
IF k1[0] OR *no writemask*:
    DEST.fp16[0] := MAX(SRC1.fp16[0], SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[j] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMAXSH __m128h _mm_mask_max_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int sae);
VMAXSH __m128h _mm_maskz_max_round_sh (__mmask8 k, __m128h a, __m128h b, int sae);
VMAXSH __m128h _mm_max_round_sh (__m128h a, __m128h b, int sae);
VMAXSH __m128h _mm_mask_max_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMAXSH __m128h _mm_maskz_max_sh (__mmask8 k, __m128h a, __m128h b);
VMAXSH __m128h _mm_max_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VMINPH—Return Minimum of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5D /r VMINPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Return the minimum packed FP16 values between xmm2 and xmm3/m128/m16bcst and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5D /r VMINPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Return the minimum packed FP16 values between ymm2 and ymm3/m256/m16bcst and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5D /r VMINPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Return the minimum packed FP16 values between zmm2 and zmm3/m512/m16bcst and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD compare of the packed FP16 values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMINPH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcast from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

```
def MIN(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 < SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2
```

VMINPH dest, src1, src2

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            tsrc2 := SRC2.fp16[0]
        ELSE:
            tsrc2 := SRC2.fp16[j]
        DEST.fp16[j] := MIN(SRC1.fp16[j], tsrc2)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINPH __m128h _mm_mask_min_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMINPH __m128h _mm_maskz_min_ph (__mmask8 k, __m128h a, __m128h b);
VMINPH __m128h _mm_min_ph (__m128h a, __m128h b);
VMINPH __m256h _mm256_mask_min_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VMINPH __m256h _mm256_maskz_min_ph (__mmask16 k, __m256h a, __m256h b);
VMINPH __m256h _mm256_min_ph (__m256h a, __m256h b);
VMINPH __m512h _mm512_mask_min_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VMINPH __m512h _mm512_maskz_min_ph (__mmask32 k, __m512h a, __m512h b);
VMINPH __m512h _mm512_min_ph (__m512h a, __m512h b);
VMINPH __m512h _mm512_mask_min_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int sae);
VMINPH __m512h _mm512_maskz_min_round_ph (__mmask32 k, __m512h a, __m512h b, int sae);
VMINPH __m512h _mm512_min_round_ph (__m512h a, __m512h b, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VMINSH—Return Minimum Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 5D /r VMINSH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Return the minimum low FP16 value between xmm3/m16 and xmm2. Stores the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a compare of the low packed FP16 values in the first source operand and the second source operand and returns the minimum value for the pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMINSH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcast from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

```
def MIN(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 < SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2
```

VMINSH dest, src1, src2

```
IF k1[0] OR *no writemask*:
    DEST.fp16[0] := MIN(SRC1.fp16[0], SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[j] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINSH __m128h _mm_mask_min_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int sae);
VMINSH __m128h _mm_maskz_min_round_sh (__mmask8 k, __m128h a, __m128h b, int sae);
VMINSH __m128h _mm_min_round_sh (__m128h a, __m128h b, int sae);
VMINSH __m128h _mm_mask_min_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMINSH __m128h _mm_maskz_min_sh (__mmask8 k, __m128h a, __m128h b);
VMINSH __m128h _mm_min_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VMOVSH—Move Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 10 /r VMOVSH xmm1{k1}{z}, m16	A	V/V	AVX512-FP16 OR AVX10.1[1]	Move FP16 value from m16 to xmm1 subject to writemask k1.
EVEX.LLIG.F3.MAP5.W0 11 /r VMOVSH m16{k1}, xmm1	B	V/V	AVX512-FP16 OR AVX10.1[1]	Move low FP16 value from xmm1 to m16 subject to writemask k1.
EVEX.LLIG.F3.MAP5.W0 10 /r VMOVSH xmm1{k1}{z}, xmm2, xmm3	C	V/V	AVX512-FP16 OR AVX10.1[1]	Move low FP16 values from xmm3 to xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].
EVEX.LLIG.F3.MAP5.W0 11 /r VMOVSH xmm1{k1}{z}, xmm2, xmm3	D	V/V	AVX512-FP16 OR AVX10.1[1]	Move low FP16 values from xmm3 to xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
D	N/A	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	N/A

Description

This instruction moves a FP16 value to a register or memory location.

The two register-only forms are aliases and differ only in where their operands are encoded; this is a side effect of the encodings selected.

Operation

VMOVSH dest, src (two operand load)

```
IF k1[0] or no writemask:
    DEST.fp16[0] := SRC.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// ELSE DEST.fp16[0] remains unchanged

DEST[MAXVL:16] := 0
```

VMOVSH dest, src (two operand store)

```
IF k1[0] or no writemask:
    DEST.fp16[0] := SRC.fp16[0]
// ELSE DEST.fp16[0] remains unchanged
```

```
VMOVSH dest, src1, src2 (three operand copy)
IF k1[0] or no writemask:
    DEST.fp16[0] := SRC2.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// ELSE DEST.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVSH __m128h _mm_load_sh (void const* mem_addr);
VMOVSH __m128h _mm_mask_load_sh (__m128h src, __mmask8 k, void const* mem_addr);
VMOVSH __m128h _mm_maskz_load_sh (__mmask8 k, void const* mem_addr);
VMOVSH __m128h _mm_mask_move_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMOVSH __m128h _mm_maskz_move_sh (__mmask8 k, __m128h a, __m128h b);
VMOVSH __m128h _mm_move_sh (__m128h a, __m128h b);
VMOVSH void _mm_mask_store_sh (void * mem_addr, __mmask8 k, __m128h a);
VMOVSH void _mm_store_sh (void * mem_addr, __m128h a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-51, “Type E5 Class Exception Conditions.”

VMOVW—Move Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.WIG 6E /r VMOVW xmm1, reg/m16	A	V/V	AVX512-FP16 OR AVX10.1[1]	Copy word from reg/m16 to xmm1.
EVEX.128.66.MAP5.WIG 7E /r VMOVW reg/m16, xmm1	B	V/V	AVX512-FP16 OR AVX10.1[1]	Copy word from xmm1 to reg/m16.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction either (a) copies one word element from an XMM register to a general-purpose register or memory location or (b) copies one word element from a general-purpose register or memory location to an XMM register. When writing a general-purpose register, the lower 16-bits of the register will contain the word value. The upper bits of the general-purpose register are written with zeros.

Operation

VMOVW dest, src (two operand load)

```
DEST.word[0] := SRC.word[0]
DEST[MAXVL:16] := 0
```

VMOVW dest, src (two operand store)

```
DEST.word[0] := SRC.word[0]
// upper bits of GPR DEST are zeroed
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVW short _mm_cvtsi128_si16 (__m128i a);
VMOVW __m128i _mm_cvtsi16_si128 (short a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Table 2-57, “Type E9NF Class Exception Conditions.”

VMULPH—Multiply Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 59 /r VMULPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from xmm3/m128/m16bcst to xmm2 and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 59 /r VMULPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Multiply packed FP16 values from ymm3/m256/m16bcst to ymm2 and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 59 /r VMULPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply packed FP16 values in zmm3/m512/m16bcst with zmm2 and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction multiplies packed FP16 values from source operands and stores the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

Operation

VMULPH (EVEX encoded versions) when src2 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.fp16[j] := SRC1.fp16[j] * SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VMULPH (EVEX encoded versions) when src2 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            DEST.fp16[j] := SRC1.fp16[j] * SRC2.fp16[0]
        ELSE:
            DEST.fp16[j] := SRC1.fp16[j] * SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULPH __m128h _mm_mask_mul_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMULPH __m128h _mm_maskz_mul_ph (__mmask8 k, __m128h a, __m128h b);
VMULPH __m128h _mm_mul_ph (__m128h a, __m128h b);
VMULPH __m256h _mm256_mask_mul_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VMULPH __m256h _mm256_maskz_mul_ph (__mmask16 k, __m256h a, __m256h b);
VMULPH __m256h _mm256_mul_ph (__m256h a, __m256h b);
VMULPH __m512h _mm512_mask_mul_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VMULPH __m512h _mm512_maskz_mul_ph (__mmask32 k, __m512h a, __m512h b);
VMULPH __m512h _mm512_mul_ph (__m512h a, __m512h b);
VMULPH __m512h _mm512_mask_mul_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VMULPH __m512h _mm512_maskz_mul_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);
VMULPH __m512h _mm512_mul_round_ph (__m512h a, __m512h b, int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

VMULSH—Multiply Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 59 /r VMULSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Multiply the low FP16 value in xmm3/m16 by low FP16 value in xmm2, and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction multiplies the low FP16 value from the source operands and stores the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VMULSH (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC2 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := SRC1.fp16[0] * SRC2.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMULSH __m128h _mm_mask_mul_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
VMULSH __m128h _mm_maskz_mul_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
VMULSH __m128h _mm_mul_round_sh (__m128h a, __m128h b, int rounding);
VMULSH __m128h _mm_mask_mul_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMULSH __m128h _mm_maskz_mul_sh (__mmask8 k, __m128h a, __m128h b);
VMULSH __m128h _mm_mul_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	Blend word integer vector zmm2 and word vector zmm3/m512 and store the result in zmm1, under control mask.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an element-by-element blending of byte/word elements between the first source operand byte vector register and the second source operand byte vector from memory or register, using the instruction mask as selector. The result is written into the destination byte vector register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit memory location.

The mask is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source, 1 for second source).

Operation

VPBLENDMB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC2[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN DEST[i+7:i] := SRC1[i+7:i]
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0;
```

VPBLENDMW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC2[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN DEST[i+15:i] := SRC1[i+15:i]
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPBLENDMB __m512i _mm512_mask_blend_epi8(__mmask64 m, __m512i a, __m512i b);
VPBLENDMB __m256i _mm256_mask_blend_epi8(__mmask32 m, __m256i a, __m256i b);
VPBLENDMB __m128i _mm_mask_blend_epi8(__mmask16 m, __m128i a, __m128i b);
VPBLENDMW __m512i _mm512_mask_blend_epi16(__mmask32 m, __m512i a, __m512i b);
VPBLENDMW __m256i _mm256_mask_blend_epi16(__mmask16 m, __m256i a, __m256i b);
VPBLENDMW __m128i _mm_mask_blend_epi16(__mmask8 m, __m128i a, __m128i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, "Type E4 Class Exception Conditions."

VPBLENDMD/VPBLENDMQ—Blend Int32/Int64 Vectors Using an OpMask Control

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Blend quadword integer vector zmm2 and quadword vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an element-by-element blending of dword/qword elements between the first source operand (the second operand) and the elements of the second source operand (the third operand) using an opmask register as select control. The blended result is written into the destination.

The destination and first source operands are ZMM registers. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for the first source operand, 1 for the second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

Operation

VPBLENDMD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN DEST[i+31:i] := SRC1[i+31:i]
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0;
```

VPBLENDMD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN DEST[i+31:i] := SRC1[i+31:i]
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDMD __m512i _mm512_mask_blend_epi32(__mmask16 k, __m512i a, __m512i b);
VPBLENDMD __m256i _mm256_mask_blend_epi32(__mmask8 m, __m256i a, __m256i b);
VPBLENDMD __m128i _mm_mask_blend_epi32(__mmask8 m, __m128i a, __m128i b);
VPBLENDMQ __m512i _mm512_mask_blend_epi64(__mmask8 k, __m512i a, __m512i b);
VPBLENDMQ __m256i _mm256_mask_blend_epi64(__mmask8 m, __m256i a, __m256i b);
VPBLENDMQ __m128i _mm_mask_blend_epi64(__mmask8 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”

VPBROADCASTB/W/D/Q—Load With Broadcast Integer Data From General Purpose Register

Opcode/ 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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	Broadcast a 32-bit value from a GPR to all doublewords in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7C /r VPBROADCASTD ymm1 {k1}{z}, r32	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Broadcast a 32-bit value from a GPR to all doublewords in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7C /r VPBROADCASTD zmm1 {k1}{z}, r32	A	V/V	AVX512F OR AVX10.1[1]	Broadcast a 32-bit value from a GPR to all doublewords in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W1 7C /r VPBROADCASTQ xmm1 {k1}{z}, r64	A	V/N.E.[1]	(AVX512VL AND AVX512F) OR AVX10.1[1]	Broadcast a 64-bit value from a GPR to all quadwords in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W1 7C /r VPBROADCASTQ ymm1 {k1}{z}, r64	A	V/N.E.[1]	(AVX512VL AND AVX512F) OR AVX10.1[1]	Broadcast a 64-bit value from a GPR to all quadwords in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W1 7C /r VPBROADCASTQ zmm1 {k1}{z}, r64	A	V/N.E.[2]	AVX512F OR AVX10.1[1]	Broadcast a 64-bit value from a GPR to all quadwords in the 512-bit destination subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.
2. EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Broadcasts a 8-bit, 16-bit, 32-bit or 64-bit value from a general-purpose register (the second operand) to all the locations in the destination vector register (the first operand) using the writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPBROADCASTB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC[7:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC[15:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[31:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[63:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPBROADCASTB __m512i _mm512_mask_set1_epi8(__m512i s, __mmask64 k, int a);
VPBROADCASTB __m512i _mm512_maskz_set1_epi8( __mmask64 k, int a);
VPBROADCASTB __m256i _mm256_mask_set1_epi8(__m256i s, __mmask32 k, int a);
VPBROADCASTB __m256i _mm256_maskz_set1_epi8( __mmask32 k, int a);
VPBROADCASTB __m128i _mm_mask_set1_epi8(__m128i s, __mmask16 k, int a);
VPBROADCASTB __m128i _mm_maskz_set1_epi8( __mmask16 k, int a);
VPBROADCASTD __m512i _mm512_mask_set1_epi32(__m512i s, __mmask16 k, 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 _mm_mask_set1_epi32(__m128i s, __mmask8 k, int a);
VPBROADCASTD __m128i _mm_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 _mm_mask_set1_epi64(__m128i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m128i _mm_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 _mm_mask_set1_epi16(__m128i s, __mmask8 k, int a);
VPBROADCASTW __m128i _mm_maskz_set1_epi16( __mmask8 k, int a);
```

Exceptions

EVEX-encoded instructions, see Table 2-55, “Type E7NM Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPBROADCAST—Load Integer and Broadcast

Opcode/ 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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	Broadcast a qword element 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.W1 59 /r VPBROADCASTQ ymm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Broadcast a qword element in source operand to locations in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 59 /r VBROADCASTI32x2 xmm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Broadcast two dword elements in source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 59 /r VBROADCASTI32x2 ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	Broadcast 256 bits of 4 quadword integer data in mem to locations in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Tuple2	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Tuple4	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
E	Tuple8	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Load integer data from the source operand (the second operand) and broadcast to all elements of the destination operand (the first operand).

VEX256-encoded VPBROADCASTB/W/D/Q: The source operand is 8-bit, 16-bit, 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. The destination operand is a YMM register. VPBROADCASTI128 support the source operand of 128-bit memory location. Register source encodings for VPBROADCASTI128 is reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX-encoded VPBROADCASTD/Q: The source operand is a 32-bit, 64-bit memory location or the low 32-bit, 64-bit data in an XMM register. The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1.

VPBROADCASTI32X4 and VPBROADCASTI64X4: The destination operand is a ZMM register and updated according to the writemask k1. The source operand is 128-bit or 256-bit memory location. Register source encodings for VBROADCASTI32X4 and VBROADCASTI64X4 are reserved and will #UD.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VPBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

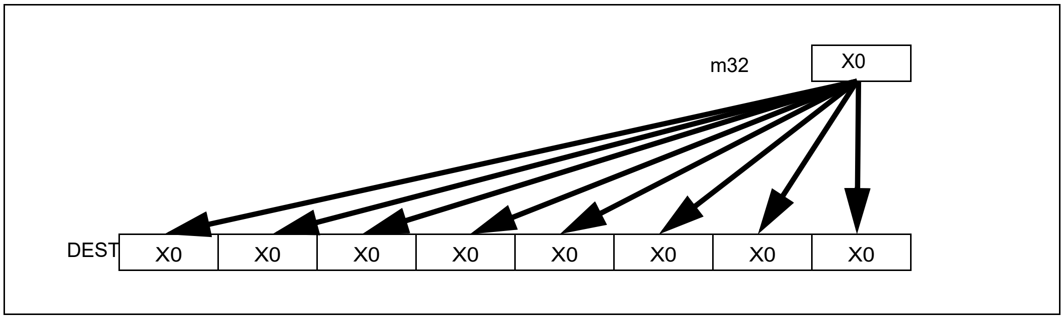


Figure 1-43. VPBROADCASTD Operation (VEX.256 encoded version)

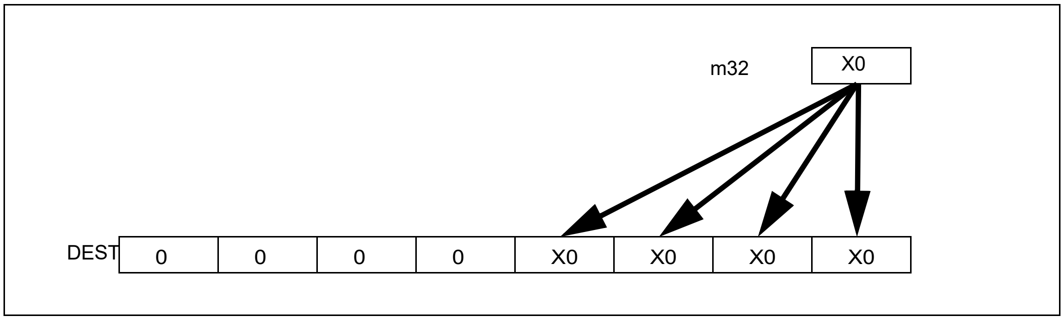


Figure 1-44. VPBROADCASTD Operation (128-bit version)

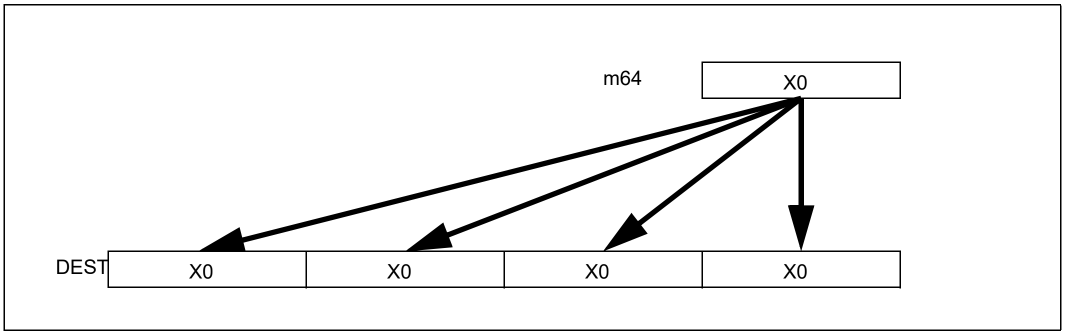


Figure 1-45. VPBROADCASTQ Operation (256-bit version)

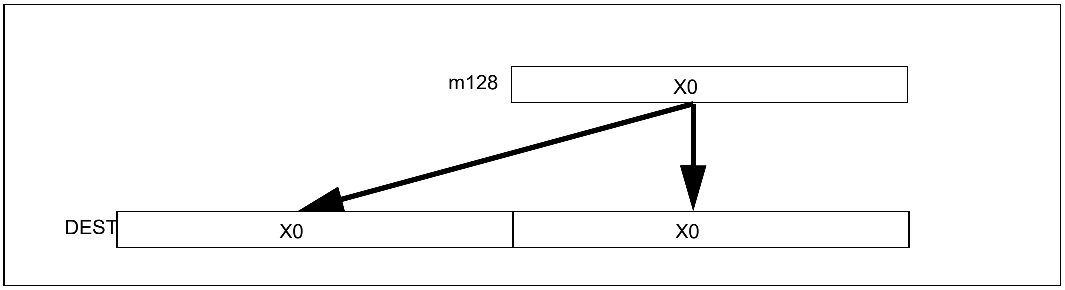


Figure 1-46. VBROADCASTI128 Operation (256-bit version)

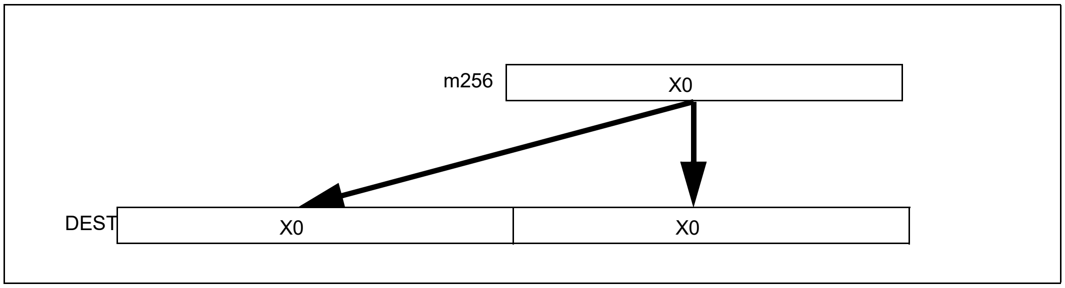


Figure 1-47. VBROADCASTI256 Operation (512-bit version)

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_i32x2(__m128i a);

VBROADCASTI32x2 __m512i _mm512_mask_broadcast_i32x2(__m512i s, __mmask16 k, __m128i a);
VBROADCASTI32x2 __m512i _mm512_maskz_broadcast_i32x2(__mmask16 k, __m128i a);
VBROADCASTI32x2 __m256i _mm256_broadcast_i32x2(__m128i a);
VBROADCASTI32x2 __m256i _mm256_mask_broadcast_i32x2(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI32x2 __m256i _mm256_maskz_broadcast_i32x2(__mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i _mm_broadcast_i32x2(__m128i a);
VBROADCASTI32x2 __m128i _mm_mask_broadcast_i32x2(__m128i s, __mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i _mm_maskz_broadcast_i32x2(__mmask8 k, __m128i a);
VBROADCASTI32x4 __m512i _mm512_broadcast_i32x4(__m128i a);
VBROADCASTI32x4 __m512i _mm512_mask_broadcast_i32x4(__m512i s, __mmask16 k, __m128i a);
VBROADCASTI32x4 __m512i _mm512_maskz_broadcast_i32x4(__mmask16 k, __m128i a);
VBROADCASTI32x4 __m256i _mm256_broadcast_i32x4(__m128i a);
VBROADCASTI32x4 __m256i _mm256_mask_broadcast_i32x4(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI32x4 __m256i _mm256_maskz_broadcast_i32x4(__mmask8 k, __m128i a);
VBROADCASTI32x8 __m512i _mm512_broadcast_i32x8(__m256i a);
VBROADCASTI32x8 __m512i _mm512_mask_broadcast_i32x8(__m512i s, __mmask16 k, __m256i a);
VBROADCASTI32x8 __m512i _mm512_maskz_broadcast_i32x8(__mmask16 k, __m256i a);
VBROADCASTI64x2 __m512i _mm512_broadcast_i64x2(__m128i a);
VBROADCASTI64x2 __m512i _mm512_mask_broadcast_i64x2(__m512i s, __mmask8 k, __m128i a);
VBROADCASTI64x2 __m512i _mm512_maskz_broadcast_i64x2(__mmask8 k, __m128i a);
VBROADCASTI64x2 __m256i _mm256_broadcast_i64x2(__m128i a);
VBROADCASTI64x2 __m256i _mm256_mask_broadcast_i64x2(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI64x2 __m256i _mm256_maskz_broadcast_i64x2(__mmask8 k, __m128i a);
VBROADCASTI64x4 __m512i _mm512_broadcast_i64x4(__m256i a);
VBROADCASTI64x4 __m512i _mm512_mask_broadcast_i64x4(__m512i s, __mmask8 k, __m256i a);
VBROADCASTI64x4 __m512i _mm512_maskz_broadcast_i64x4(__mmask8 k, __m256i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Table 2-23, “Type 6 Class Exception Conditions.”
EVEX-encoded instructions, syntax with reg/mem operand, see Table 2-53, “Type E6 Class Exception Conditions.”
Additionally:

#UD	If VEX.L = 0 for VPBROADCASTQ, VPBROADCASTI128.
	If EVEX.L’L = 0 for VBROADCASTI32X4/VBROADCASTI64X2.
	If EVEX.L’L < 10b for VBROADCASTI32X8/VBROADCASTI64X4.

VPBROADCASTM—Broadcast Mask to Vector Register

Opcode/ 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 AND AVX512CD) OR AVX10.1[1]	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 AND AVX512CD) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512CD) OR AVX10.1[1]	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 AND AVX512CD) OR AVX10.1[1]	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 OR AVX10.1[1]	Broadcast low word value in k1 to sixteen locations in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Broadcasts the zero-extended 64/32 bit value of the low byte/word of the source operand (the second operand) to each 64/32 bit element of the destination operand (the first operand). The source operand is an opmask register. The destination operand is a ZMM register (EVEX.512), YMM register (EVEX.256), or XMM register (EVEX.128).

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPBROADCASTMB2Q

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j*64
    DEST[i+63:i] := ZeroExtend(SRC[7:0])
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTMW2D

```
(KL, VL) = (4, 128), (8, 256),(16, 512)
FOR j := 0 TO KL-1
    i := j*32
    DEST[i+31:i] := ZeroExtend(SRC[15:0])
ENDFOR
```

```
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPBROADCASTMB2Q __m512i _mm512_broadcastmb_epi64( __mmask8);
VPBROADCASTMW2D __m512i _mm512_broadcastmw_epi32( __mmask16);
VPBROADCASTMB2Q __m256i _mm256_broadcastmb_epi64( __mmask8);
VPBROADCASTMW2D __m256i _mm256_broadcastmw_epi32( __mmask8);
VPBROADCASTMB2Q __m128i _mm_broadcastmb_epi64( __mmask8);
VPBROADCASTMW2D __m128i _mm_broadcastmw_epi32( __mmask8);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-54, "Type E6NF Class Exception Conditions."

VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 3F /r ib VPCMPB k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed byte values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPB performs a comparison between pairs of signed byte values.

VPCMPUB performs a comparison between pairs of unsigned byte values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 64/32/16 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 1-18.

Table 1-18. Pseudo-Op and VPCMP* Implementation

Pseudo-Op	**PCMPM Implementation**
VPCMPEQ* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 0*
VPCMPLT* *reg1, reg2, reg3*	VPCMP**reg1, reg2, reg3, 1*
VPCMPLE* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 2*
VPCMPNEQ* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 4*
VPPCMPNLT* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 5*
VPCMPNLE* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 6*

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 in Table 2-49, “Type E4 Class Exception Conditions.”

VPCMPD/VPCMPUD—Compare Packed Integer Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

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 1-18.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP := EQ;
    1: OP := LT;
    2: OP := LE;
    3: OP := FALSE;
    4: OP := NEQ;
    5: OP := NLT;
    6: OP := NLE;
    7: OP := TRUE;
ESAC;
```

VPCMPD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+31:i] OP SRC2[31:0];
                ELSE CMP := SRC1[i+31:i] OP SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPCMPUD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+31:i] OP SRC2[31:0];
                ELSE CMP := SRC1[i+31:i] OP SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPD __mmask16 _mm512_cmp_epi32_mask(__m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_mask_cmp_epi32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m512i a, __m512i b);
VPCMPD __mmask16 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPUD __mmask16 _mm512_cmp_epu32_mask(__m512i a, __m512i b, int imm);
VPCMPUD __mmask16 _mm512_mask_cmp_epu32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPUD __mmask16 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__m512i a, __m512i b);
VPCMPUD __mmask16 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPD __mmask8 _mm256_cmp_epi32_mask(__m256i a, __m256i b, int imm);
VPCMPD __mmask8 _mm256_mask_cmp_epi32_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPD __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m256i a, __m256i b);
VPCMPD __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPUD __mmask8 _mm256_cmp_epu32_mask(__m256i a, __m256i b, int imm);
VPCMPUD __mmask8 _mm256_mask_cmp_epu32_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPUD __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__m256i a, __m256i b);
VPCMPUD __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPD __mmask8 _mm_cmp_epi32_mask(__m128i a, __m128i b, int imm);
VPCMPD __mmask8 _mm_mask_cmp_epi32_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPD __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m128i a, __m128i b);
VPCMPD __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPUD __mmask8 _mm_cmp_epu32_mask(__m128i a, __m128i b, int imm);
VPCMPUD __mmask8 _mm_mask_cmp_epu32_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPUD __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__m128i a, __m128i b);
VPCMPUD __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

VPCMPQ/VPCMPUQ—Compare Packed Integer Values Into Mask

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

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 1-18.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP := EQ;
    1: OP := LT;
    2: OP := LE;
    3: OP := FALSE;
    4: OP := NEQ;
    5: OP := NLT;
    6: OP := NLE;
    7: OP := TRUE;
ESAC;
```

VPCMPQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+63:i] OP SRC2[63:0];
                ELSE CMP := SRC1[i+63:i] OP SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPCMPUQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+63:i] OP SRC2[63:0];
                ELSE CMP := SRC1[i+63:i] OP SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPQ __mmask8 _mm512_cmp_epi64_mask(__m512i a, __m512i b, int imm);
VPCMPQ __mmask8 _mm512_mask_cmp_epi64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__m512i a, __m512i b);
VPCMPQ __mmask8 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPUQ __mmask8 _mm512_cmp_epu64_mask(__m512i a, __m512i b, int imm);
VPCMPUQ __mmask8 _mm512_mask_cmp_epu64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPUQ __mmask8 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__m512i a, __m512i b);
VPCMPUQ __mmask8 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPQ __mmask8 _mm256_cmp_epi64_mask(__m256i a, __m256i b, int imm);
VPCMPQ __mmask8 _mm256_mask_cmp_epi64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPQ __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__m256i a, __m256i b);
VPCMPQ __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPUQ __mmask8 _mm256_cmp_epu64_mask(__m256i a, __m256i b, int imm);
VPCMPUQ __mmask8 _mm256_mask_cmp_epu64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPUQ __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__m256i a, __m256i b);
VPCMPUQ __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPQ __mmask8 _mm_cmp_epi64_mask(__m128i a, __m128i b, int imm);
VPCMPQ __mmask8 _mm_mask_cmp_epi64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPQ __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__m128i a, __m128i b);
VPCMPQ __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPUQ __mmask8 _mm_cmp_epu64_mask(__m128i a, __m128i b, int imm);
VPCMPUQ __mmask8 _mm_mask_cmp_epu64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPUQ __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__m128i a, __m128i b);
VPCMPUQ __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask

Opcode/ 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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	Compare packed unsigned word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W1 3E /r ib VPCMPUW k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed integer word in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPW performs a comparison between pairs of signed word values.

VPCMPUW performs a comparison between pairs of unsigned word values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 32/16/8 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 1-18.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP := EQ;
    1: OP := LT;
    2: OP := LE;
    3: OP := FALSE;
    4: OP := NEQ;
    5: OP := NLT;
    6: OP := NLE;
    7: OP := TRUE;
ESAC;
```

VPCMPW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k2[j] OR *no writemask*
        THEN
            ICMP := SRC1[i+15:i] OP SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] = 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPCMPUW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k2[j] OR *no writemask*
        THEN
            CMP := SRC1[i+15:i] OP SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] = 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPW __mmask32 _mm512_cmp_epi16_mask(__m512i a, __m512i b, int cmp);
VPCMPW __mmask32 _mm512_mask_cmp_epi16_mask(__mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPW __mmask16 _mm256_cmp_epi16_mask(__m256i a, __m256i b, int cmp);
VPCMPW __mmask16 _mm256_mask_cmp_epi16_mask(__mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPW __mmask8 _mm_cmp_epi16_mask(__m128i a, __m128i b, int cmp);
VPCMPW __mmask8 _mm_mask_cmp_epi16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPW __mmask32 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__m512i a, __m512i b);
VPCMPW __mmask32 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask32 m, __m512i a, __m512i b);
VPCMPW __mmask16 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__m256i a, __m256i b);
VPCMPW __mmask16 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask16 m, __m256i a, __m256i b);
VPCMPW __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__m128i a, __m128i b);
VPCMPW __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask8 m, __m128i a, __m128i b);
VPCMPUW __mmask32 _mm512_cmp_epu16_mask(__m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 _mm512_mask_cmp_epu16_mask(__mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 _mm256_cmp_epu16_mask(__m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 _mm256_mask_cmp_epu16_mask(__mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 _mm_cmp_epu16_mask(__m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 _mm_mask_cmp_epu16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPUW __mmask32 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

VPCOMPRESSB/VCOMPRESSW—Store Sparse Packed Byte/Word Integer Values Into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 63 /r VPCOMPRESSB m128{k1}, xmm1	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Compress up to 128 bits of packed byte values from xmm1 to m128 with writemask k1.
EVEX.128.66.0F38.W0 63 /r VPCOMPRESSB xmm1{k1}{z}, xmm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Compress up to 128 bits of packed byte values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W0 63 /r VPCOMPRESSB m256{k1}, ymm1	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Compress up to 256 bits of packed byte values from ymm1 to m256 with writemask k1.
EVEX.256.66.0F38.W0 63 /r VPCOMPRESSB ymm1{k1}{z}, ymm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Compress up to 256 bits of packed byte values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W0 63 /r VPCOMPRESSB m512{k1}, zmm1	A	V/V	AVX512_VBMI2 OR AVX10.1[1]	Compress up to 512 bits of packed byte values from zmm1 to m512 with writemask k1.
EVEX.512.66.0F38.W0 63 /r VPCOMPRESSB zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2 OR AVX10.1[1]	Compress up to 512 bits of packed byte values from zmm2 to zmm1 with writemask k1.
EVEX.128.66.0F38.W1 63 /r VPCOMPRESSW m128{k1}, xmm1	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Compress up to 128 bits of packed word values from xmm1 to m128 with writemask k1.
EVEX.128.66.0F38.W1 63 /r VPCOMPRESSW xmm1{k1}{z}, xmm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Compress up to 128 bits of packed word values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W1 63 /r VPCOMPRESSW m256{k1}, ymm1	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Compress up to 256 bits of packed word values from ymm1 to m256 with writemask k1.
EVEX.256.66.0F38.W1 63 /r VPCOMPRESSW ymm1{k1}{z}, ymm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Compress up to 256 bits of packed word values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W1 63 /r VPCOMPRESSW m512{k1}, zmm1	A	V/V	AVX512_VBMI2 OR AVX10.1[1]	Compress up to 512 bits of packed word values from zmm1 to m512 with writemask k1.
EVEX.512.66.0F38.W1 63 /r VPCOMPRESSW zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2 OR AVX10.1[1]	Compress up to 512 bits of packed word values from zmm2 to zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Compress (stores) up to 64 byte values or 32 word values from the source operand (second operand) to the destination operand (first operand), based on the active elements determined by the writemask operand. Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Moves up to 512 bits of packed byte values from the source operand (second operand) to the destination operand (first operand). This instruction is used to store partial contents of a vector register into a byte vector or single memory location using the active elements in operand writemask.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

This instruction supports memory fault suppression.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPCOMPRESSB store form

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[k] := SRC.byte[j]
        k := k +1
```

VPCOMPRESSB reg-reg form

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[k] := SRC.byte[j]
        k := k + 1
IF *merging-masking*:
    *DEST[VL-1:k*8] remains unchanged*
    ELSE DEST[VL-1:k*8] := 0
DEST[MAX_VL-1:VL] := 0
```

VPCOMPRESSW store form

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.word[k] := SRC.word[j]
        k := k + 1
```

VPCOMPRESSW reg-reg form

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.word[k] := SRC.word[j]
        k := k + 1
IF *merging-masking*:
    *DEST[VL-1:k*16] remains unchanged*
    ELSE DEST[VL-1:k*16] := 0
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPCOMPRESSB __m128i _mm_mask_compress_epi8(__m128i, __mmask16, __m128i);
VPCOMPRESSB __m128i _mm_maskz_compress_epi8(__mmask16, __m128i);
VPCOMPRESSB __m256i _mm256_mask_compress_epi8(__m256i, __mmask32, __m256i);
VPCOMPRESSB __m256i _mm256_maskz_compress_epi8(__mmask32, __m256i);
VPCOMPRESSB __m512i _mm512_mask_compress_epi8(__m512i, __mmask64, __m512i);
VPCOMPRESSB __m512i _mm512_maskz_compress_epi8(__mmask64, __m512i);
VPCOMPRESSB  void _mm_mask_compressstoreu_epi8(void*, __mmask16, __m128i);
VPCOMPRESSB  void _mm256_mask_compressstoreu_epi8(void*, __mmask32, __m256i);
VPCOMPRESSB  void _mm512_mask_compressstoreu_epi8(void*, __mmask64, __m512i);
VPCOMPRESSW  __m128i  _mm_mask_compress_epi16(__m128i, __mmask8, __m128i);
VPCOMPRESSW  __m128i  _mm_maskz_compress_epi16(__mmask8, __m128i);
VPCOMPRESSW  __m256i  _mm256_mask_compress_epi16(__m256i, __mmask16, __m256i);
VPCOMPRESSW  __m256i  _mm256_maskz_compress_epi16(__mmask16, __m256i);
VPCOMPRESSW  __m512i  _mm512_mask_compress_epi16(__m512i, __mmask32, __m512i);
VPCOMPRESSW  __m512i  _mm512_maskz_compress_epi16(__mmask32, __m512i);
VPCOMPRESSW  void  _mm_mask_compressstoreu_epi16(void*, __mmask8, __m128i);
VPCOMPRESSW  void  _mm256_mask_compressstoreu_epi16(void*, __mmask16, __m256i);
VPCOMPRESSW  void  _mm512_mask_compressstoreu_epi16(void*, __mmask32, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”

VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values Into Dense Memory/Register

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	Compress packed doubleword integer values from xmm2 to xmm1/m128 using control mask k1.
EVEX.256.66.0F38.W0 8B /r VPCOMPRESSD ymm1/m256 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compress packed doubleword integer values from ymm2 to ymm1/m256 using control mask k1.
EVEX.512.66.0F38.W0 8B /r VPCOMPRESSD zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Compress packed doubleword integer values from zmm2 to zmm1/m512 using control mask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Compress (store) up to 16/8/4 doubleword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPCOMPRESSD (EVEX encoded versions) store form

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE := 32
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+31:i]
            k := k + SIZE
    FI;
```

```
ENDFOR;
```

VPCOMPRESSD (EVEX encoded versions) reg-reg form

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE := 32
k := 0
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no controlmask*
       THEN
           DEST[k+SIZE-1:k] := SRC[i+31:i]
           k := k + SIZE
   FI;
ENDFOR
IF *merging-masking*
       THEN *DEST[VL-1:k] remains unchanged*
       ELSE DEST[VL-1:k] := 0
FI
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPCOMPRESSD __m512i _mm512_mask_compress_epi32(__m512i s, __mmask16 c, __m512i a);
VPCOMPRESSD __m512i _mm512_maskz_compress_epi32( __mmask16 c, __m512i a);
VPCOMPRESSD void _mm512_mask_compressstoreu_epi32(void * a, __mmask16 c, __m512i s);
VPCOMPRESSD __m256i _mm256_mask_compress_epi32(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSD __m256i _mm256_maskz_compress_epi32( __mmask8 c, __m256i a);
VPCOMPRESSD void _mm256_mask_compressstoreu_epi32(void * a, __mmask8 c, __m256i s);
VPCOMPRESSD __m128i _mm_mask_compress_epi32(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSD __m128i _mm_maskz_compress_epi32( __mmask8 c, __m128i a);
VPCOMPRESSD void _mm_mask_compressstoreu_epi32(void * a, __mmask8 c, __m128i s);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values Into Dense Memory/Register

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	Compress packed quadword integer values from xmm2 to xmm1/m128 using control mask k1.
EVEX.256.66.0F38.W1 8B /r VPCOMPRESSQ ymm1/m256 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Compress packed quadword integer values from ymm2 to ymm1/m256 using control mask k1.
EVEX.512.66.0F38.W1 8B /r VPCOMPRESSQ zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Compress packed quadword integer values from zmm2 to zmm1/m512 using control mask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Compress (stores) up to 8/4/2 quadword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPCOMPRESSQ (EVEX encoded versions) store form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE := 64
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+63:i]
            k := k + SIZE
    FI;
```

```
ENFOR
```

VPCOMPRESSQ (EVEX encoded versions) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE := 64
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+63:i]
            k := k + SIZE
    FI;
ENDFOR
IF *merging-masking*
        THEN *DEST[VL-1:k] remains unchanged*
        ELSE DEST[VL-1:k] := 0
FI
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCOMPRESSQ __m512i _mm512_mask_compress_epi64(__m512i s, __mmask8 c, __m512i a);
VPCOMPRESSQ __m512i _mm512_maskz_compress_epi64(__mmask8 c, __m512i a);
VPCOMPRESSQ void _mm512_mask_compressstoreu_epi64(void * a, __mmask8 c, __m512i s);
VPCOMPRESSQ __m256i _mm256_mask_compress_epi64(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSQ __m256i _mm256_maskz_compress_epi64(__mmask8 c, __m256i a);
VPCOMPRESSQ void _mm256_mask_compressstoreu_epi64(void * a, __mmask8 c, __m256i s);
VPCOMPRESSQ __m128i _mm_mask_compress_epi64(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSQ __m128i _mm_maskz_compress_epi64(__mmask8 c, __m128i a);
VPCOMPRESSQ void _mm_mask_compressstoreu_epi64(void * a, __mmask8 c, __m128i s);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

VPCONFLICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values Into Dense Memory/ Register

Opcode/ 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 AND AVX512CD) OR AVX10.1[1]	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 AND AVX512CD) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512CD) OR AVX10.1[1]	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 AND AVX512CD) OR AVX10.1[1]	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 OR AVX10.1[1]	Detect duplicate quad-word values in zmm2/m512/m64bcst using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Test each dword/qword element of the source operand (the second operand) for equality with all other elements in the source operand closer to the least significant element. Each element’s comparison results form a bit vector, which is then zero extended and written to the destination according to the writemask.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPCONFLICTD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j*32
    IF MaskBit(j) OR *no writemask*THEN
        FOR k := 0 TO j-1
            m := k*32
            IF ((SRC[i+31:i] = SRC[m+31:m])) THEN
                DEST[i+k] := 1
            ELSE
                DEST[i+k] := 0
            FI
        ENDFOR
        DEST[i+31:i+j] := 0
    ELSE
        IF *merging-masking* THEN
            *DEST[i+31:i] remains unchanged*
        ELSE
            DEST[i+31:i] := 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPCONFLICTQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j*64
    IF MaskBit(j) OR *no writemask*THEN
        FOR k := 0 TO j-1
            m := k*64
            IF ((SRC[i+63:i] = SRC[m+63:m])) THEN
                DEST[i+k] := 1
            ELSE
                DEST[i+k] := 0
            FI
        ENDFOR
        DEST[i+63:i+j] := 0
    ELSE
        IF *merging-masking* THEN
            *DEST[i+63:i] remains unchanged*
        ELSE
            DEST[i+63:i] := 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCONFLICTD __m512i _mm512_conflict_epi32(__m512i a);
VPCONFLICTD __m512i _mm512_mask_conflict_epi32(__m512i s, __mmask16 m, __m512i a);
VPCONFLICTD __m512i _mm512_maskz_conflict_epi32(__mmask16 m, __m512i a);
VPCONFLICTQ __m512i _mm512_conflict_epi64(__m512i a);
VPCONFLICTQ __m512i _mm512_mask_conflict_epi64(__m512i s, __mmask8 m, __m512i a);
VPCONFLICTQ __m512i _mm512_maskz_conflict_epi64(__mmask8 m, __m512i a);
VPCONFLICTD __m256i _mm256_conflict_epi32(__m256i a);
VPCONFLICTD __m256i _mm256_mask_conflict_epi32(__m256i s, __mmask8 m, __m256i a);
VPCONFLICTD __m256i _mm256_maskz_conflict_epi32(__mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_conflict_epi64(__m256i a);
VPCONFLICTQ __m256i _mm256_mask_conflict_epi64(__m256i s, __mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_maskz_conflict_epi64(__mmask8 m, __m256i a);
VPCONFLICTD __m128i _mm_conflict_epi32(__m128i a);
VPCONFLICTD __m128i _mm_mask_conflict_epi32(__m128i s, __mmask8 m, __m128i a);
VPCONFLICTD __m128i _mm_maskz_conflict_epi32(__mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_conflict_epi64(__m128i a);
VPCONFLICTQ __m128i _mm_mask_conflict_epi64(__m128i s, __mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_maskz_conflict_epi64(__mmask8 m, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”

VPDPBUSD—Multiply and Add Unsigned and Signed Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 50 /r VPDPBUSD xmm1, xmm2, xmm3/m128	A	V/V	AVX-VNNI	Multiply groups of 4 pairs of signed bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1.
VEX.256.66.0F38.W0 50 /r VPDPBUSD ymm1, ymm2, ymm3/m256	A	V/V	AVX-VNNI	Multiply groups of 4 pairs of signed bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1.
EVEX.128.66.0F38.W0 50 /r VPDPBUSD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1[1]	Multiply groups of 4 pairs of signed bytes in xmm3/m128/m32bcst with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 50 /r VPDPBUSD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1[1]	Multiply groups of 4 pairs of signed bytes in ymm3/m256/m32bcst with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 50 /r VPDPBUSD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VNNI OR AVX10.1[1]	Multiply groups of 4 pairs of signed bytes in zmm3/m512/m32bcst with corresponding unsigned bytes of zmm2, summing those products and adding them to doubleword result in zmm1 under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual unsigned bytes of the first source operand by the corresponding signed bytes of the second source operand, producing intermediate signed word results. The word results are then summed and accumulated in the destination dword element size operand.

This instruction supports memory fault suppression.

Operation

VPDPBUSD dest, src1, src2 (VEX encoded versions)

```
VL=(128, 256)
KL=VL/32

ORIGDEST := DEST
FOR i := 0 TO KL-1:
```

```
    // Extending to 16b
    // src1extend := ZERO_EXTEND
    // src2extend := SIGN_EXTEND

    p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.byte[4*i+0])
    p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
    p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
    p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])
    DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word

DEST[MAX_VL-1:VL] := 0
```

VPDPBUSD dest, src1, src2 (EVEX encoded versions)

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    IF k1[i] or *no writemask*:
        // Byte elements of SRC1 are zero-extended to 16b and
        // byte elements of SRC2 are sign extended to 16b before multiplication.
        IF SRC2 is memory and EVEX.b == 1:
            t := SRC2.dword[0]
        ELSE:
            t := SRC2.dword[i]
        p1word := ZERO_EXTEND(SRC1.byte[4*i]) * SIGN_EXTEND(t.byte[0])
        p2word := ZERO_EXTEND(SRC1.byte[4*i+1]) * SIGN_EXTEND(t.byte[1])
        p3word := ZERO_EXTEND(SRC1.byte[4*i+2]) * SIGN_EXTEND(t.byte[2])
        p4word := ZERO_EXTEND(SRC1.byte[4*i+3]) * SIGN_EXTEND(t.byte[3])
        DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE:       // Merge masking, dest element unchanged
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPDPBUSD __m128i _mm_dpbusd_avx_epi32(__m128i, __m128i, __m128i);
VPDPBUSD __m128i _mm_dpbusd_epi32(__m128i, __m128i, __m128i);
VPDPBUSD __m128i _mm_mask_dpbusd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPBUSD __m128i _mm_maskz_dpbusd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPBUSD __m256i _mm256_dpbusd_avx_epi32(__m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_dpbusd_epi32(__m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_mask_dpbusd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPBUSD __m256i _mm256_maskz_dpbusd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPBUSD __m512i _mm512_dpbusd_epi32(__m512i, __m512i, __m512i);
VPDPBUSD __m512i _mm512_mask_dpbusd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPBUSD __m512i _mm512_maskz_dpbusd_epi32(__mmask16, __m512i, __m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

VPDPBUSDS—Multiply and Add Unsigned and Signed Bytes With Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 51 /r VPDPBUSDS xmm1, xmm2, xmm3/m128	A	V/V	AVX-VNNI	Multiply groups of 4 pairs signed bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result, with signed saturation in xmm1.
VEX.256.66.0F38.W0 51 /r VPDPBUSDS ymm1, ymm2, ymm3/m256	A	V/V	AVX-VNNI	Multiply groups of 4 pairs signed bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result, with signed saturation in ymm1.
EVEX.128.66.0F38.W0 51 /r VPDPBUSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1[1]	Multiply groups of 4 pairs signed bytes in xmm3/m128/m32bcst with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result, with signed saturation in xmm1, under writemask k1.
EVEX.256.66.0F38.W0 51 /r VPDPBUSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1[1]	Multiply groups of 4 pairs signed bytes in ymm3/m256/m32bcst with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result, with signed saturation in ymm1, under writemask k1.
EVEX.512.66.0F38.W0 51 /r VPDPBUSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VNNI OR AVX10.1[1]	Multiply groups of 4 pairs signed bytes in zmm3/m512/m32bcst with corresponding unsigned bytes of zmm2, summing those products and adding them to doubleword result, with signed saturation in zmm1, under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual unsigned bytes of the first source operand by the corresponding signed bytes of the second source operand, producing intermediate signed word results. The word results are then summed and accumulated in the destination dword element size operand. If the intermediate sum overflows a 32b signed number the result is saturated to either 0x7FFF_FFFF for positive numbers of 0x8000_0000 for negative numbers.

This instruction supports memory fault suppression.

Operation

VPDPBUSDS dest, src1, src2 (VEX encoded versions)

```
VL=(128, 256)
KL=VL/32

ORIGDEST := DEST
FOR i := 0 TO KL-1:
    // Extending to 16b
    // src1extend := ZERO_EXTEND
    // src2extend := SIGN_EXTEND

    p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.byte[4*i+0])
    p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
    p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
    p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])
    DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)

DEST[MAX_VL-1:VL] := 0
```

VPDPBUSDS dest, src1, src2 (EVEX encoded versions)

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    IF k1[i] or *no writemask*:
        // Byte elements of SRC1 are zero-extended to 16b and
        // byte elements of SRC2 are sign extended to 16b before multiplication.
        IF SRC2 is memory and EVEX.b == 1:
            t := SRC2.dword[0]
        ELSE:
            t := SRC2.dword[i]
        p1word := ZERO_EXTEND(SRC1.byte[4*i]) * SIGN_EXTEND(t.byte[0])
        p2word := ZERO_EXTEND(SRC1.byte[4*i+1]) * SIGN_EXTEND(t.byte[1])
        p3word := ZERO_EXTEND(SRC1.byte[4*i+2]) * SIGN_EXTEND(t.byte[2])
        p4word := ZERO_EXTEND(SRC1.byte[4*i+3]) *SIGN_EXTEND(t.byte[3])
        DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE:      // Merge masking, dest element unchanged
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPDPBUSDS __m128i _mm_dpbusds_avx_epi32(__m128i, __m128i, __m128i);
VPDPBUSDS __m128i _mm_dpbusds_epi32(__m128i, __m128i, __m128i);
VPDPBUSDS __m128i _mm_mask_dpbusds_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPBUSDS __m128i _mm_maskz_dpbusds_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPBUSDS __m256i _mm256_dpbusds_avx_epi32(__m256i, __m256i, __m256i);
VPDPBUSDS __m256i _mm256_dpbusds_epi32(__m256i, __m256i, __m256i);
VPDPBUSDS __m256i _mm256_mask_dpbusds_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPBUSDS __m256i _mm256_maskz_dpbusds_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPBUSDS __m512i _mm512_dpbusds_epi32(__m512i, __m512i, __m512i);
VPDPBUSDS __m512i _mm512_mask_dpbusds_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPBUSDS __m512i _mm512_maskz_dpbusds_epi32(__mmask16, __m512i, __m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

VPDPWSSD—Multiply and Add Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 52 /r VPDPWSSD xmm1, xmm2, xmm3/m128	A	V/V	AVX-VNNI	Multiply groups of 2 pairs signed words in xmm3/m128 with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1.
VEX.256.66.0F38.W0 52 /r VPDPWSSD ymm1, ymm2, ymm3/m256	A	V/V	AVX-VNNI	Multiply groups of 2 pairs signed words in ymm3/m256 with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1.
EVEX.128.66.0F38.W0 52 /r VPDPWSSD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1[1]	Multiply groups of 2 pairs signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, under writemask k1.
EVEX.256.66.0F38.W0 52 /r VPDPWSSD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1[1]	Multiply groups of 2 pairs signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, under writemask k1.
EVEX.512.66.0F38.W0 52 /r VPDPWSSD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VNNI OR AVX10.1[1]	Multiply groups of 2 pairs signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result in zmm1, under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand.

This instruction supports memory fault suppression.

Operation

VPDPWSSD dest, src1, src2 (VEX encoded versions)

```
VL=(128, 256)
KL=VL/32
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    p1dword := SIGN_EXTEND(SRC1.word[2*i+0]) * SIGN_EXTEND(SRC2.word[2*i+0] )
    p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(SRC2.word[2*i+1] )
```

```
    DEST.dword[i] := ORIGDEST.dword[i] + p1dword + p2dword
DEST[MAX_VL-1:VL] := 0
```

VPDPWSSD dest, src1, src2 (EVEX encoded versions)

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and EVEX.b == 1:
            t := SRC2.dword[0]
        ELSE:
            t := SRC2.dword[i]
        p1dword := SIGN_EXTEND(SRC1.word[2*i]) * SIGN_EXTEND(t.word[0])
        p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(t.word[1])
        DEST.dword[i] := ORIGDEST.dword[i] + p1dword + p2dword
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE:       // Merge masking, dest element unchanged
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPDPWSSD __m128i _mm_dpwssd_avx_epi32(__m128i, __m128i, __m128i);
VPDPWSSD __m128i _mm_dpwssd_epi32(__m128i, __m128i, __m128i);
VPDPWSSD __m128i _mm_mask_dpwssd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPWSSD __m128i _mm_maskz_dpwssd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPWSSD __m256i _mm256_dpwssd_avx_epi32(__m256i, __m256i, __m256i);
VPDPWSSD __m256i _mm256_dpwssd_epi32(__m256i, __m256i, __m256i);
VPDPWSSD __m256i _mm256_mask_dpwssd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPWSSD __m256i _mm256_maskz_dpwssd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPWSSD __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i);
VPDPWSSD __m512i _mm512_mask_dpwssd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPWSSD __m512i _mm512_maskz_dpwssd_epi32(__mmask16, __m512i, __m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

VPDPWSSDS—Multiply and Add Signed Word Integers With Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 53 /r VPDPWSSDS xmm1, xmm2, xmm3/m128	A	V/V	AVX-VNNI	Multiply groups of 2 pairs of signed words in xmm3/m128 with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, with signed saturation.
VEX.256.66.0F38.W0 53 /r VPDPWSSDS ymm1, ymm2, ymm3/m256	A	V/V	AVX-VNNI	Multiply groups of 2 pairs of signed words in ymm3/m256 with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, with signed saturation.
EVEX.128.66.0F38.W0 53 /r VPDPWSSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1[1]	Multiply groups of 2 pairs of signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, with signed saturation, under writemask k1.
EVEX.256.66.0F38.W0 53 /r VPDPWSSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1[1]	Multiply groups of 2 pairs of signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, with signed saturation, under writemask k1.
EVEX.512.66.0F38.W0 53 /r VPDPWSSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VNNI OR AVX10.1[1]	Multiply groups of 2 pairs of signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result in zmm1, with signed saturation, under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand. If the intermediate sum overflows a 32b signed number, the result is saturated to either 0x7FFF_FFFF for positive numbers of 0x8000_0000 for negative numbers.

This instruction supports memory fault suppression.

Operation

VPDPWSSDS dest, src1, src2 (VEX encoded versions)

```
VL=(128, 256)
KL=VL/32
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    p1dword := SIGN_EXTEND(SRC1.word[2*i+0]) * SIGN_EXTEND(SRC2.word[2*i+0])
    p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(SRC2.word[2*i+1])
    DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
DEST[MAX_VL-1:VL] := 0
```

VPDPWSSDS dest, src1, src2 (EVEX encoded versions)

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and EVEX.b == 1:
            t := SRC2.dword[0]
        ELSE:
            t := SRC2.dword[i]
        p1dword := SIGN_EXTEND(SRC1.word[2*i]) * SIGN_EXTEND(t.word[0])
        p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(t.word[1])
        DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE:       // Merge masking, dest element unchanged
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPDPWSSDS __m128i _mm_dpwssds_avx_epi32(__m128i, __m128i, __m128i);
VPDPWSSDS __m128i _mm_dpwssds_epi32(__m128i, __m128i, __m128i);
VPDPWSSDS __m128i _mm_mask_dpwssd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPWSSDS __m128i _mm_maskz_dpwssd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPWSSDS __m256i _mm256_dpwssds_avx_epi32(__m256i, __m256i, __m256i);
VPDPWSSDS __m256i _mm256_dpwssd_epi32(__m256i, __m256i, __m256i);
VPDPWSSDS __m256i _mm256_mask_dpwssd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPWSSDS __m256i _mm256_maskz_dpwssd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPWSSDS __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i);
VPDPWSSDS __m512i _mm512_mask_dpwssd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPWSSDS __m512i _mm512_maskz_dpwssd_epi32(__mmask16, __m512i, __m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

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 AND AVX512_VBMI) OR AVX10.1[1]	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) OR AVX10.1[1]	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 OR AVX10.1[1]	Permute bytes in zmm3/m512 using byte indexes in zmm2 and store the result in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Copies bytes from the second source operand (the third operand) to the destination operand (the first operand) according to the byte indices in the first source operand (the second operand). Note that this instruction permits a byte in the source operand to be copied to more than one location in the destination operand.

Only the low 6(EVEX.512)/5(EVEX.256)/4(EVEX.128) bits of each byte index is used to select the location of the source byte from the second source operand.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated at byte granularity by the writemask k1.

Operation

VPERMB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128:
    n := 3;
ELSE IF VL = 256:
    n := 4;
ELSE IF VL = 512:
    n := 5;
FI;
FOR j := 0 TO KL-1:
    id := SRC1[j*8 + n : j*8] ; // location of the source byte
    IF k1[j] OR *no writemask* THEN
        DEST[j*8 + 7: j*8] := SRC2[id*8 +7: id*8];
    ELSE IF zeroing-masking THEN
        DEST[j*8 + 7: j*8] := 0;
    *ELSE
        DEST[j*8 + 7: j*8] remains unchanged*
    FI
```

```
ENDFOR
DEST[MAX_VL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMB __m512i _mm512_permutexvar_epi8( __m512i idx, __m512i a);
VPERMB __m512i _mm512_mask_permutexvar_epi8(__m512i s, __mmask64 k, __m512i idx, __m512i a);
VPERMB __m512i _mm512_maskz_permutexvar_epi8( __mmask64 k, __m512i idx, __m512i a);
VPERMB __m256i _mm256_permutexvar_epi8( __m256i idx, __m256i a);
VPERMB __m256i _mm256_mask_permutexvar_epi8(__m256i s, __mmask32 k, __m256i idx, __m256i a);
VPERMB __m256i _mm256_maskz_permutexvar_epi8( __mmask32 k, __m256i idx, __m256i a);
VPERMB __m128i _mm_permutexvar_epi8( __m128i idx, __m128i a);
VPERMB __m128i _mm_mask_permutexvar_epi8(__m128i s, __mmask16 k, __m128i idx, __m128i a);
VPERMB __m128i _mm_maskz_permutexvar_epi8( __mmask16 k, __m128i idx, __m128i a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

VPERMD/VPERMW—Permute Packed Doubleword/Word Elements

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	Permute word integers in zmm3/m512 using indexes in zmm2 and store the result in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Copies doublewords (or words) from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword (word) in the source operand to be copied to more than one location in the destination operand.

VEX.256 encoded VPERMD: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPERMD: The first and second operands are ZMM/YMM registers, the third operand can be a ZMM/YMM register, a 512/256-bit memory location or a 512/256-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

VPERMW: first and second operands are ZMM/YMM/XMM registers, the third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination is updated using the writemask k1.

EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

Operation

VPERMD (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
IF VL = 256 THEN n := 2; FI;
IF VL = 512 THEN n := 3; FI;
FOR j := 0 TO KL-1
    i := j * 32
    id := 32*SRC1[i+n:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := SRC2[31:0];
                ELSE DEST[i+31:i] := SRC2[id+31:id];
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMD (VEX.256 encoded version)

```
DEST[31:0] := (SRC2[255:0] >> (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
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMD __m512i _mm512_permutexvar_epi32(__m512i idx, __m512i a);
VPERMD __m512i _mm512_mask_permutexvar_epi32(__m512i s, __mmask16 k, __m512i idx, __m512i a);
VPERMD __m512i _mm512_maskz_permutexvar_epi32(__mmask16 k, __m512i idx, __m512i a);
VPERMD __m256i _mm256_permutexvar_epi32(__m256i idx, __m256i a);
VPERMD __m256i _mm256_mask_permutexvar_epi32(__m256i s, __mmask8 k, __m256i idx, __m256i a);
VPERMD __m256i _mm256_maskz_permutexvar_epi32(__mmask8 k, __m256i idx, __m256i a);
VPERMW __m512i _mm512_permutexvar_epi16(__m512i idx, __m512i a);
VPERMW __m512i _mm512_mask_permutexvar_epi16(__m512i s, __mmask32 k, __m512i idx, __m512i a);
VPERMW __m512i _mm512_maskz_permutexvar_epi16(__mmask32 k, __m512i idx, __m512i a);
VPERMW __m256i _mm256_permutexvar_epi16(__m256i idx, __m256i a);
VPERMW __m256i _mm256_mask_permutexvar_epi16(__m256i s, __mmask16 k, __m256i idx, __m256i a);
VPERMW __m256i _mm256_maskz_permutexvar_epi16(__mmask16 k, __m256i idx, __m256i a);
VPERMW __m128i _mm_permutexvar_epi16(__m128i idx, __m128i a);
VPERMW __m128i _mm_mask_permutexvar_epi16(__m128i s, __mmask8 k, __m128i idx, __m128i a);
VPERMW __m128i _mm_maskz_permutexvar_epi16(__mmask8 k, __m128i idx, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPERMD, see Table 2-50, “Type E4NF Class Exception Conditions.”
EVEX-encoded VPERMW, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
Additionally:

#UD	If VEX.L = 0. If EVEX.L’L = 0 for VPERMD.

VPERMI2B—Full Permute of Bytes From Two Tables Overwriting the Index

Opcode/ 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 AND AVX512_VBMI) OR AVX10.1[1]	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) OR AVX10.1[1]	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 OR AVX10.1[1]	Permute bytes in zmm3/m512 and zmm2 using byte indexes in zmm1 and store the byte results in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Permutes byte values in the second operand (the first source operand) and the third operand (the second source operand) using the byte indices in the first operand (the destination operand) to select byte elements from the second or third source operands. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result. The third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the same tables can be reused in subsequent iterations, but the index elements are overwritten.

Bits (MAX_VL-1:256/128) of the destination are zeroed for VL=256,128.

Operation

VPERMI2B (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128:
    id := 3;
ELSE IF VL = 256:
    id := 4;
ELSE IF VL = 512:
    id := 5;
FI;
TMP_DEST[VL-1:0] := DEST[VL-1:0];
FOR j := 0 TO KL-1
    off := 8*SRC1[j*8 + id: j*8] ;
    IF k1[j] OR *no writemask*:
        DEST[j*8 + 7: j*8] := TMP_DEST[j*8+id+1]? SRC2[off+7:off] : SRC1[off+7:off];
    ELSE IF *zeroing-masking*
        DEST[j*8 + 7: j*8] := 0;
    *ELSE
        DEST[j*8 + 7: j*8] remains unchanged*
    FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMI2B __m512i _mm512_permutex2var_epi8(__m512i a, __m512i idx, __m512i b);
VPERMI2B __m512i _mm512_mask2_permutex2var_epi8(__m512i a, __m512i idx, __mmask64 k, __m512i b);
VPERMI2B __m512i _mm512_maskz_permutex2var_epi8(__mmask64 k, __m512i a, __m512i idx, __m512i b);
VPERMI2B __m256i _mm256_permutex2var_epi8(__m256i a, __m256i idx, __m256i b);
VPERMI2B __m256i _mm256_mask2_permutex2var_epi8(__m256i a, __m256i idx, __mmask32 k, __m256i b);
VPERMI2B __m256i _mm256_maskz_permutex2var_epi8(__mmask32 k, __m256i a, __m256i idx, __m256i b);
VPERMI2B __m128i _mm_permutex2var_epi8(__m128i a, __m128i idx, __m128i b);
VPERMI2B __m128i _mm_mask2_permutex2var_epi8(__m128i a, __m128i idx, __mmask16 k, __m128i b);
VPERMI2B __m128i _mm_maskz_permutex2var_epi8(__mmask16 k, __m128i a, __m128i idx, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

VPERMI2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index

Opcode/ 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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	Permute single-precision floating-point values from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 77 /r VPERMI2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Permute single-precision floating-point values from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 77 /r VPERMI2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1[1]	Permute single-precision floating-point values from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	Permute double precision floating-point values from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 77 /r VPERMI2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Permute double precision floating-point values from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 77 /r VPERMI2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1[1]	Permute double precision floating-point values from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r,w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

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[i+31:i] remains unchanged*
                    ELSE                                ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMI2Q/VPERMI2PD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8 512)
IF VL = 128
    id := 0
FI;
IF VL = 256
    id := 1
FI;
IF VL = 512
    id := 2
FI;
TMP_DEST:= DEST
FOR j := 0 TO KL-1
    i := j * 64
    off := 64*TMP_DEST[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := TMP_DEST[i+id+1] ? SRC2[63:0]
                    : SRC1[off+63:off]
            ELSE
                DEST[i+63:i] := TMP_DEST[i+id+1] ? SRC2[off+63:off]
                    : SRC1[off+63:off]
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMI2D __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERMI2D __m512i _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMI2D __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMI2D __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERMI __m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERMI2D __m256i _mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMI2D __m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMI2D __m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMI2D __m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERMI2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMI2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512i idx, __m512d b);
VPERMI2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d b);
VPERMI2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d b);
VPERMI2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d b);
VPERMI2PD __m256d _mm256_permutex2var_pd(__m256d a, __m256i idx, __m256d b);
VPERMI2PD __m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256i idx, __m256d b);
VPERMI2PD __m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256i idx, __mmask8 k, __m256d b);
VPERMI2PD __m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256i idx, __m256d b);
VPERMI2PD __m128d _mm_permutex2var_pd(__m128d a, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b);
VPERMI2PD __m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128i idx, __m128d b);
VPERMI2PS __m512 _mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
VPERMI2PS __m512 _mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
VPERMI2PS __m512 _mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512 b);
VPERMI2PS __m512 _mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);
VPERMI2PS __m256 _mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMI2PS __m256 _mm256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b);
VPERMI2PS __m256 _mm256_mask2_permutex2var_ps(__m256 a, __m256i idx, __mmask8 k, __m256 b);
VPERMI2PS __m256 _mm256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b);
VPERMI2PS __m128 _mm_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b);
VPERMI2PS __m128 _mm_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);
VPERMI2Q __m512i _mm512_permutex2var_epi64(__m512i a, __m512i idx, __m512i b);
VPERMI2Q __m512i _mm512_mask_permutex2var_epi64(__m512i a, __mmask8 k, __m512i idx, __m512i b);
VPERMI2Q __m512i _mm512_mask2_permutex2var_epi64(__m512i a, __m512i idx, __mmask8 k, __m512i b);
VPERMI2Q __m512i _mm512_maskz_permutex2var_epi64(__mmask8 k, __m512i a, __m512i idx, __m512i b);
VPERMI2Q __m256i _mm256_permutex2var_epi64(__m256i a, __m256i idx, __m256i b);
VPERMI2Q __m256i _mm256_mask_permutex2var_epi64(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMI2Q __m256i _mm256_mask2_permutex2var_epi64(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMI2Q __m256i _mm256_maskz_permutex2var_epi64(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMI2Q __m128i _mm_permutex2var_epi64(__m128i a, __m128i idx, __m128i b);
VPERMI2Q __m128i _mm_mask_permutex2var_epi64(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2Q __m128i _mm_mask2_permutex2var_epi64(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2Q __m128i _mm_maskz_permutex2var_epi64(__mmask8 k, __m128i a, __m128i idx, __m128i b);

VPERMI2W __m512i _mm512_permutex2var_epi16(__m512i a, __m512i idx, __m512i b);
VPERMI2W __m512i _mm512_mask_permutex2var_epi16(__m512i a, __mmask32 k, __m512i idx, __m512i b);
VPERMI2W __m512i _mm512_mask2_permutex2var_epi16(__m512i a, __m512i idx, __mmask32 k, __m512i b);
VPERMI2W __m512i _mm512_maskz_permutex2var_epi16(__mmask32 k, __m512i a, __m512i idx, __m512i b);
VPERMI2W __m256i _mm256_permutex2var_epi16(__m256i a, __m256i idx, __m256i b);
VPERMI2W __m256i _mm256_mask_permutex2var_epi16(__m256i a, __mmask16 k, __m256i idx, __m256i b);
VPERMI2W __m256i _mm256_mask2_permutex2var_epi16(__m256i a, __m256i idx, __mmask16 k, __m256i b);
VPERMI2W __m256i _mm256_maskz_permutex2var_epi16(__mmask16 k, __m256i a, __m256i idx, __m256i b);
VPERMI2W __m128i _mm_permutex2var_epi16(__m128i a, __m128i idx, __m128i b);
VPERMI2W __m128i _mm_mask_permutex2var_epi16(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2W __m128i _mm_mask2_permutex2var_epi16(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2W __m128i _mm_maskz_permutex2var_epi16(__mmask8 k, __m128i a, __m128i idx, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VPERMI2D/Q/PS/PD: See Table 2-50, “Type E4NF Class Exception Conditions.”
VPERMI2W: See Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

VPERMILPD—Permute In-Lane of Pairs of Double Precision Floating-Point Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Permute double precision floating-point values in zmm2/m512/m64bcst using controls from imm8 and store the result in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

(variable control version)

Permute pairs of double precision floating-point values in the first source operand (second operand), each using a 1-bit control field residing in the corresponding quadword element of the second source operand (third operand). Permuted results are stored in the destination operand (first operand).

The control bits are located at bit 0 of each quadword element (see Figure 1-49). 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.

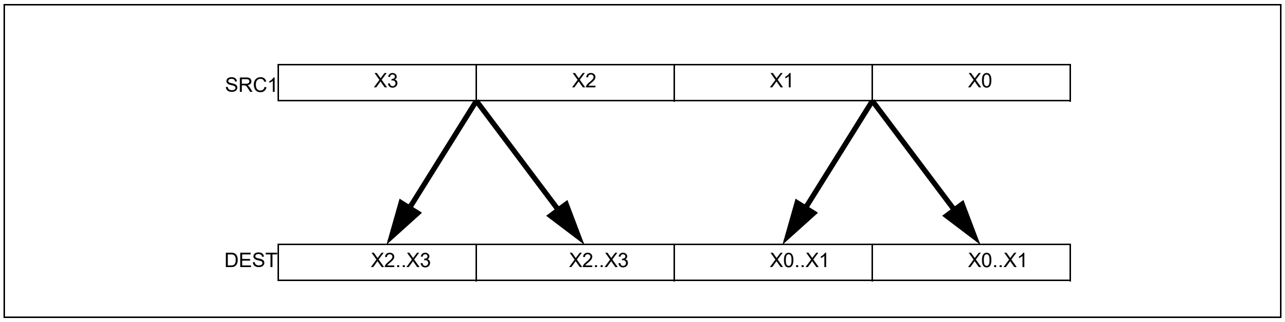


Figure 1-48. VPERMILPD Operation

VEX.256 encoded version: Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

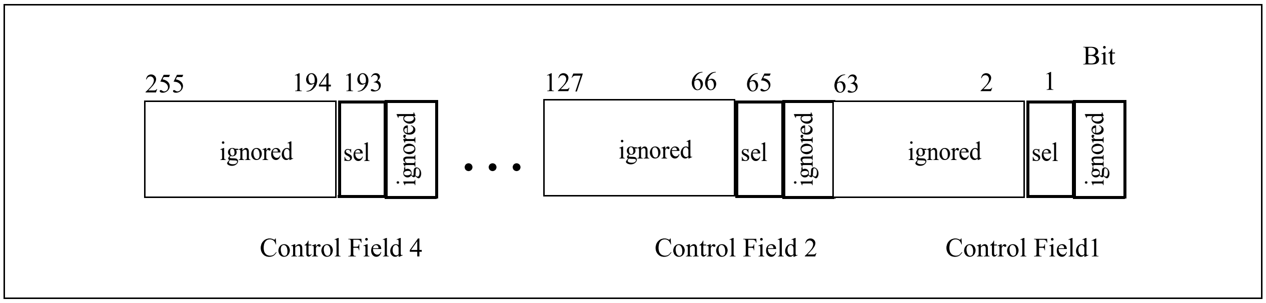


Figure 1-49. VPERMILPD Shuffle Control

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;
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 Table 2-21, “Type 4 Class Exception Conditions.”

Additionally:

#UD If VEX.W = 1.

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”

Additionally:

#UD If either (E)VEX.vvvv != 1111B and with imm8.

VPERMILPS—Permute In-Lane of Quadruples of Single Precision Floating-Point Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Permute single-precision floating-point values zmm2/m512/m32bcst using controls from imm8 and store the result in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Variable control version:

Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the corresponding dword element of the second source operand. Permuted results are stored in the destination operand (first operand).

The 2-bit control fields are located at the low two bits of each dword element (see Figure 1-51). 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.

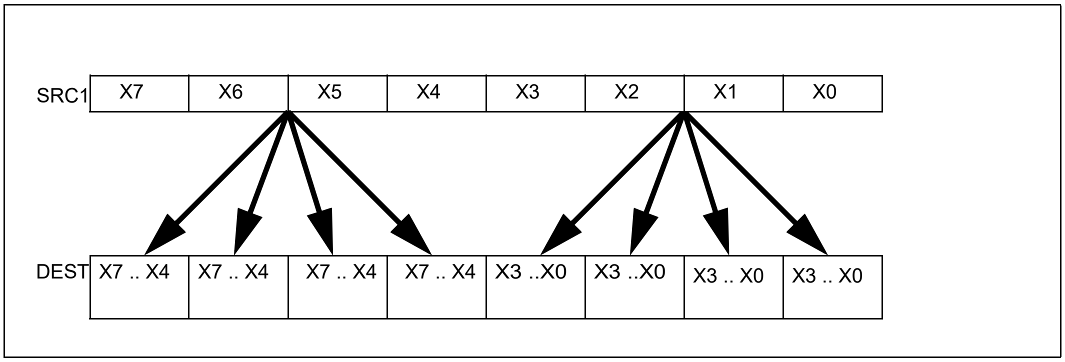


Figure 1-50. VPERMILPS Operation

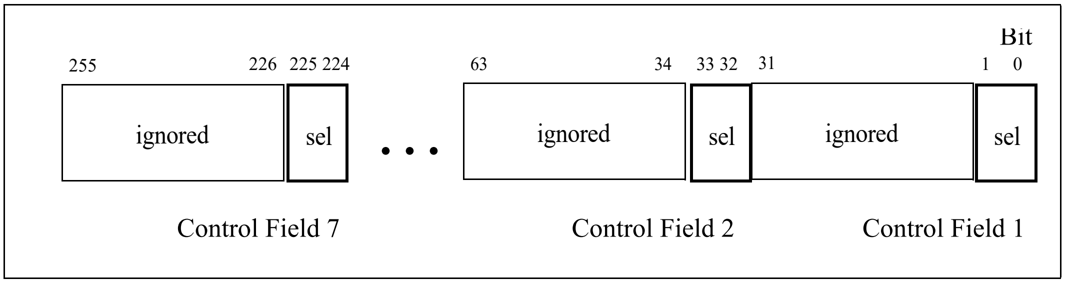


Figure 1-51. VPERMILPS Shuffle Control

(immediate control version)

Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the imm8 byte. Each 128-bit lane in the destination operand (first operand) use the four control fields of the same imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

Note: For the imm8 version, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

Operation

```
Select4(SRC, control) {
CASE (control[1:0]) OF
    0:    TMP := SRC[31:0];
    1:    TMP := SRC[63:32];
    2:    TMP := SRC[95:64];
    3:    TMP := SRC[127:96];
ESAC;
RETURN TMP
}
```

VPERMILPS (EVEX immediate versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
        THEN TMP_SRC1[i+31:i] := SRC1[31:0];
        ELSE TMP_SRC1[i+31:i] := SRC1[i+31:i];
    FI;
ENDFOR;

TMP_DEST[31:0] := Select4(TMP_SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] := Select4(TMP_SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] := Select4(TMP_SRC1[127:0], imm8[5:4]);
TMP_DEST[127:96] := Select4(TMP_SRC1[127:0], imm8[7:6]); FI;
IF VL >= 256
    TMP_DEST[159:128] := Select4(TMP_SRC1[255:128], imm8[1:0]); FI;
    TMP_DEST[191:160] := Select4(TMP_SRC1[255:128], imm8[3:2]); FI;
    TMP_DEST[223:192] := Select4(TMP_SRC1[255:128], imm8[5:4]); FI;
    TMP_DEST[255:224] := Select4(TMP_SRC1[255:128], imm8[7:6]); FI;
FI;
IF VL >= 512
    TMP_DEST[287:256] := Select4(TMP_SRC1[383:256], imm8[1:0]); FI;
    TMP_DEST[319:288] := Select4(TMP_SRC1[383:256], imm8[3:2]); FI;
    TMP_DEST[351:320] := Select4(TMP_SRC1[383:256], imm8[5:4]); FI;
    TMP_DEST[383:352] := Select4(TMP_SRC1[383:256], imm8[7:6]); FI;
    TMP_DEST[415:384] := Select4(TMP_SRC1[511:384], imm8[1:0]); FI;
    TMP_DEST[447:416] := Select4(TMP_SRC1[511:384], imm8[3:2]); FI;
    TMP_DEST[479:448] := Select4(TMP_SRC1[511:384], imm8[5:4]); FI;
    TMP_DEST[511:480] := Select4(TMP_SRC1[511:384], imm8[7:6]); FI;
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*
                THEN *DEST[i+31:i] remains unchanged*
                ELSE DEST[i+31:i] := 0                  ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMILPS (256-bit immediate version)

```
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] := Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC1[127:0], imm8[7:6]);
DEST[159:128] := Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] := Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] := Select4(SRC1[255:128], imm8[5:4]);
DEST[255:224] := Select4(SRC1[255:128], imm8[7:6]);
```

VPERMILPS (128-bit immediate version)

```
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] := Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC1[127:0], imm8[7:6]);
DEST[MAXVL-1:128] := 0
```

VPERMILPS (EVEX variable versions)

```
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0];
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i];
    FI;
ENDFOR;
TMP_DEST[31:0] := Select4(SRC1[127:0], TMP_SRC2[1:0]);
TMP_DEST[63:32] := Select4(SRC1[127:0], TMP_SRC2[33:32]);
TMP_DEST[95:64] := Select4(SRC1[127:0], TMP_SRC2[65:64]);
TMP_DEST[127:96] := Select4(SRC1[127:0], TMP_SRC2[97:96]);
IF VL >= 256
    TMP_DEST[159:128] := Select4(SRC1[255:128], TMP_SRC2[129:128]);
    TMP_DEST[191:160] := Select4(SRC1[255:128], TMP_SRC2[161:160]);
    TMP_DEST[223:192] := Select4(SRC1[255:128], TMP_SRC2[193:192]);
    TMP_DEST[255:224] := Select4(SRC1[255:128], TMP_SRC2[225:224]);
FI;
IF VL >= 512
    TMP_DEST[287:256] := Select4(SRC1[383:256], TMP_SRC2[257:256]);
    TMP_DEST[319:288] := Select4(SRC1[383:256], TMP_SRC2[289:288]);
    TMP_DEST[351:320] := Select4(SRC1[383:256], TMP_SRC2[321:320]);
    TMP_DEST[383:352] := Select4(SRC1[383:256], TMP_SRC2[353:352]);
    TMP_DEST[415:384] := Select4(SRC1[511:384], TMP_SRC2[385:384]);
    TMP_DEST[447:416] := Select4(SRC1[511:384], TMP_SRC2[417:416]);
    TMP_DEST[479:448] := Select4(SRC1[511:384], TMP_SRC2[449:448]);
    TMP_DEST[511:480] := Select4(SRC1[511:384], TMP_SRC2[481:480]);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*
                THEN *DEST[i+31:i] remains unchanged*
                ELSE DEST[i+31:i] := 0                ;zeroing-masking
```

```
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMILPS (256-bit variable version)

```
DEST[31:0] := Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] := Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] := Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] := Select4(SRC1[127:0], SRC2[97:96]);
DEST[159:128] := Select4(SRC1[255:128], SRC2[129:128]);
DEST[191:160] := Select4(SRC1[255:128], SRC2[161:160]);
DEST[223:192] := Select4(SRC1[255:128], SRC2[193:192]);
DEST[255:224] := Select4(SRC1[255:128], SRC2[225:224]);
DEST[MAXVL-1:256] := 0
```

VPERMILPS (128-bit variable version)

```
DEST[31:0] := Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] := Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] :=Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] := Select4(SRC1[127:0], SRC2[97:96]);
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMILPS __m512 _mm512_permute_ps( __m512 a, int imm);
VPERMILPS __m512 _mm512_mask_permute_ps(__m512 s, __mmask16 k, __m512 a, int imm);
VPERMILPS __m512 _mm512_maskz_permute_ps( __mmask16 k, __m512 a, int imm);
VPERMILPS __m256 _mm256_mask_permute_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 _mm256_maskz_permute_ps( __mmask8 k, __m256 a, int imm);
VPERMILPS __m128 _mm_mask_permute_ps(__m128 s, __mmask8 k, __m128 a, int imm);
VPERMILPS __m128 _mm_maskz_permute_ps( __mmask8 k, __m128 a, int imm);
VPERMILPS __m512 _mm512_permutevar_ps( __m512i i, __m512 a);
VPERMILPS __m512 _mm512_mask_permutevar_ps(__m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m512 _mm512_maskz_permutevar_ps( __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m256 _mm256_mask_permutevar_ps(__m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m256 _mm256_maskz_permutevar_ps( __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m128 _mm_mask_permutevar_ps(__m128 s, __mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 _mm_maskz_permutevar_ps( __mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 _mm_permute_ps (__m128 a, int control);
VPERMILPS __m256 _mm256_permute_ps (__m256 a, int control);
VPERMILPS __m128 _mm_permutevar_ps (__m128 a, __m128i control);
VPERMILPS __m256 _mm256_permutevar_ps (__m256 a, __m256i control);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

Additionally:

#UD	If VEX.W = 1.

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”

Additionally:

#UD	If either (E)VEX.vvvv != 1111B and with imm8.

VPERMPD—Permute Double Precision Floating-Point Elements

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Permute double precision floating-point elements in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1 subject to writemask k1.
EVEX.256.66.0F38.W1 16 /r VPERMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Permute double precision floating-point elements in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The imm8 version: Copies quadword elements of double precision floating-point values from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

The imm8 versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadword elements of double precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPD is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMPD (EVEX - imm8 control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN TMP_SRC[i+63:i] := SRC[63:0];
        ELSE TMP_SRC[i+63:i] := SRC[i+63:i];
    FI;
ENDFOR;

TMP_DEST[63:0] := (TMP_SRC[256:0] >> (IMM8[1:0] * 64))[63:0];
TMP_DEST[127:64] := (TMP_SRC[256:0] >> (IMM8[3:2] * 64))[63:0];
TMP_DEST[191:128] := (TMP_SRC[256:0] >> (IMM8[5:4] * 64))[63:0];
TMP_DEST[255:192] := (TMP_SRC[256:0] >> (IMM8[7:6] * 64))[63:0];
IF VL >= 512
    TMP_DEST[319:256] := (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
    TMP_DEST[383:320] := (TMP_SRC[511:256] >> (IMM8[3:2] * 64))[63:0];
    TMP_DEST[447:384] := (TMP_SRC[511:256] >> (IMM8[5:4] * 64))[63:0];
    TMP_DEST[511:448] := (TMP_SRC[511:256] >> (IMM8[7:6] * 64))[63:0];
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0                       ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMPD (EVEX - vector control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0];
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
    FI;
ENDFOR;

IF VL = 256
    TMP_DEST[63:0] := (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
    TMP_DEST[191:128] := (TMP_SRC2[255:0] >> (SRC1[129:128] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;
```

```
IF VL = 512
    TMP_DEST[63:0] := (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
    TMP_DEST[191:128] := (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
    TMP_DEST[319:256] := (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
    TMP_DEST[383:320] := (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
    TMP_DEST[447:384] := (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
    TMP_DEST[511:448] := (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0                   ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMPD (VEX.256 encoded version)

```
DEST[63:0] := (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] := (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] := (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] := (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMPD __m512d _mm512_permutex_pd( __m512d a, int imm);
VPERMPD __m512d _mm512_mask_permutex_pd(__m512d s, __mmask16 k, __m512d a, int imm);
VPERMPD __m512d _mm512_maskz_permutex_pd( __mmask16 k, __m512d a, int imm);
VPERMPD __m512d _mm512_permutexvar_pd( __m512i i, __m512d a);
VPERMPD __m512d _mm512_mask_permutexvar_pd(__m512d s, __mmask16 k, __m512i i, __m512d a);
VPERMPD __m512d _mm512_maskz_permutexvar_pd( __mmask16 k, __m512i i, __m512d a);
VPERMPD __m256d _mm256_permutex_epi64( __m256d a, int imm);
VPERMPD __m256d _mm256_mask_permutex_epi64(__m256i s, __mmask8 k, __m256d a, int imm);
VPERMPD __m256d _mm256_maskz_permutex_epi64( __mmask8 k, __m256d a, int imm);
VPERMPD __m256d _mm256_permutexvar_epi64( __m256i i, __m256d a);
VPERMPD __m256d _mm256_mask_permutexvar_epi64(__m256i s, __mmask8 k, __m256i i, __m256d a);
VPERMPD __m256d _mm256_maskz_permutexvar_epi64( __mmask8 k, __m256i i, __m256d a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions”; additionally:

#UD	If VEX.L = 0.
	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions”; additionally:

#UD	If encoded with EVEX.128.
	If EVEX.vvvv != 1111B and with imm8.

VPERMPS—Permute Single Precision Floating-Point Elements

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Permute single-precision floating-point values in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 subject to write mask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Copies doubleword elements of single-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword in the source operand to be copied to more than one location in the destination operand.

VEX.256 versions: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded version: The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

If VPERMPS is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMPS (EVEX forms)

```
(KL, VL) (8, 256),= (16, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0];
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i];
    FI;
ENDFOR;

IF VL = 256
    TMP_DEST[31:0] := (TMP_SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
    TMP_DEST[63:32] := (TMP_SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
```

```
    TMP_DEST[95:64] := (TMP_SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
    TMP_DEST[127:96] := (TMP_SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
    TMP_DEST[159:128] := (TMP_SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
    TMP_DEST[191:160] := (TMP_SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
    TMP_DEST[223:192] := (TMP_SRC2[255:0] >> (SRC1[193:192] * 32))[31:0];
    TMP_DEST[255:224] := (TMP_SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
FI;
IF VL = 512
    TMP_DEST[31:0] := (TMP_SRC2[511:0] >> (SRC1[3:0] * 32))[31:0];
    TMP_DEST[63:32] := (TMP_SRC2[511:0] >> (SRC1[35:32] * 32))[31:0];
    TMP_DEST[95:64] := (TMP_SRC2[511:0] >> (SRC1[67:64] * 32))[31:0];
    TMP_DEST[127:96] := (TMP_SRC2[511:0] >> (SRC1[99:96] * 32))[31:0];
    TMP_DEST[159:128] := (TMP_SRC2[511:0] >> (SRC1[131:128] * 32))[31:0];
    TMP_DEST[191:160] := (TMP_SRC2[511:0] >> (SRC1[163:160] * 32))[31:0];
    TMP_DEST[223:192] := (TMP_SRC2[511:0] >> (SRC1[195:192] * 32))[31:0];
    TMP_DEST[255:224] := (TMP_SRC2[511:0] >> (SRC1[227:224] * 32))[31:0];
    TMP_DEST[287:256] := (TMP_SRC2[511:0] >> (SRC1[259:256] * 32))[31:0];
    TMP_DEST[319:288] := (TMP_SRC2[511:0] >> (SRC1[291:288] * 32))[31:0];
    TMP_DEST[351:320] := (TMP_SRC2[511:0] >> (SRC1[323:320] * 32))[31:0];
    TMP_DEST[383:352] := (TMP_SRC2[511:0] >> (SRC1[355:352] * 32))[31:0];
    TMP_DEST[415:384] := (TMP_SRC2[511:0] >> (SRC1[387:384] * 32))[31:0];
    TMP_DEST[447:416] := (TMP_SRC2[511:0] >> (SRC1[419:416] * 32))[31:0];
    TMP_DEST[479:448] :=(TMP_SRC2[511:0] >> (SRC1[451:448] * 32))[31:0];
    TMP_DEST[511:480] := (TMP_SRC2[511:0] >> (SRC1[483:480] * 32))[31:0];
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0                   ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMPS (VEX.256 encoded version)

```
DEST[31:0] := (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] := (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] := (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] := (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] := (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] := (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] := (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] := (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAXVL-1:256] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMPS __m512 _mm512_permutexvar_ps(__m512i i, __m512 a);
VPERMPS __m512 _mm512_mask_permutexvar_ps(__m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMPS __m512 _mm512_maskz_permutexvar_ps(__mmask16 k, __m512i i, __m512 a);
VPERMPS __m256 _mm256_permutexvar_ps(__m256 i, __m256 a);
VPERMPS __m256 _mm256_mask_permutexvar_ps(__m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMPS __m256 _mm256_maskz_permutexvar_ps(__mmask8 k, __m256 i, __m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”

VPERMQ—Qwords Element Permutation

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Permute qwords in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The imm8 version: Copies quadwords from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

Immediate control versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadwords from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPQ is encoded with VEX.L= 0 or EVEX.128, an attempt to execute the instruction will cause an #UD exception.

Operation

VPERMQ (EVEX - imm8 control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN TMP_SRC[i+63:i] := SRC[63:0];
        ELSE TMP_SRC[i+63:i] := SRC[i+63:i];
    FI;
ENDFOR;
    TMP_DEST[63:0] := (TMP_SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
    TMP_DEST[191:128] := (TMP_SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
IF VL >= 512
    TMP_DEST[319:256] := (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
    TMP_DEST[383:320] := (TMP_SRC[511:256] >> (IMM8[3:2] * 64))[63:0];
    TMP_DEST[447:384] := (TMP_SRC[511:256] >> (IMM8[5:4] * 64))[63:0];
    TMP_DEST[511:448] := (TMP_SRC[511:256] >> (IMM8[7:6] * 64))[63:0];
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0                   ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMQ (EVEX - vector control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0];
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[63:0] := (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
    TMP_DEST[191:128] := (TMP_SRC2[255:0] >> (SRC1[129:128] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;
IF VL = 512
    TMP_DEST[63:0] := (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
```

```
    TMP_DEST[127:64] := (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
    TMP_DEST[191:128] := (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
    TMP_DEST[319:256] := (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
    TMP_DEST[383:320] := (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
    TMP_DEST[447:384] := (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
    TMP_DEST[511:448] := (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0                   ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMQ (VEX.256 encoded version)

```
DEST[63:0] := (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] := (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] := (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] := (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMQ __m512i _mm512_permutex_epi64( __m512i a, int imm);
VPERMQ __m512i _mm512_mask_permutex_epi64(__m512i s, __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_maskz_permutex_epi64( __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_permutexvar_epi64( __m512i a, __m512i b);
VPERMQ __m512i _mm512_mask_permutexvar_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPERMQ __m512i _mm512_maskz_permutexvar_epi64( __mmask8 k, __m512i a, __m512i b);
VPERMQ __m256i _mm256_permutex_epi64( __m256i a, int imm);
VPERMQ __m256i _mm256_mask_permutex_epi64(__m256i s, __mmask8 k, __m256i a, int imm);
VPERMQ __m256i _mm256_maskz_permutex_epi64( __mmask8 k, __m256i a, int imm);
VPERMQ __m256i _mm256_permutexvar_epi64( __m256i a, __m256i b);
VPERMQ __m256i _mm256_mask_permutexvar_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPERMQ __m256i _mm256_maskz_permutexvar_epi64( __mmask8 k, __m256i a, __m256i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 0.
	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”

Additionally:

#UD	If encoded with EVEX.128.
	If EVEX.vvvv != 1111B and with imm8.

VPERMT2B—Full Permute of Bytes From Two Tables Overwriting a Table

Opcode/ 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 AND AVX512_VBMI) OR AVX10.1[1]	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) OR AVX10.1[1]	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 OR AVX10.1[1]	Permute bytes in zmm3/m512 and zmm1 using byte indexes in zmm2 and store the byte results in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Permutes byte values from two tables, comprising of the first operand (also the destination operand) and the third operand (the second source operand). The second operand (the first source operand) provides byte indices to select byte results from the two tables. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result. The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the second table and the indices can be reused in subsequent iterations, but the first table is overwritten.

Bits (MAX_VL-1:256/128) of the destination are zeroed for VL=256,128.

Operation

VPERMT2B (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128:
    id := 3;
ELSE IF VL = 256:
    id := 4;
ELSE IF VL = 512:
    id := 5;
FI;
TMP_DEST[VL-1:0] := DEST[VL-1:0];
FOR j := 0 TO KL-1
    off := 8*SRC1[j*8 + id: j*8] ;
    IF k1[j] OR *no writemask*:
        DEST[j*8 + 7: j*8] := SRC1[j*8+id+1]? SRC2[off+7:off] : TMP_DEST[off+7:off];
    ELSE IF *zeroing-masking*
        DEST[j*8 + 7: j*8] := 0;
    *ELSE
        DEST[j*8 + 7: j*8] remains unchanged*
    FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMT2B __m512i _mm512_permutex2var_epi8(__m512i a, __m512i idx, __m512i b);
VPERMT2B __m512i _mm512_mask_permutex2var_epi8(__m512i a, __mmask64 k, __m512i idx, __m512i b);
VPERMT2B __m512i _mm512_maskz_permutex2var_epi8(__mmask64 k, __m512i a, __m512i idx, __m512i b);
VPERMT2B __m256i _mm256_permutex2var_epi8(__m256i a, __m256i idx, __m256i b);
VPERMT2B __m256i _mm256_mask_permutex2var_epi8(__m256i a, __mmask32 k, __m256i idx, __m256i b);
VPERMT2B __m256i _mm256_maskz_permutex2var_epi8(__mmask32 k, __m256i a, __m256i idx, __m256i b);
VPERMT2B __m128i _mm_permutex2var_epi8(__m128i a, __m128i idx, __m128i b);
VPERMT2B __m128i _mm_mask_permutex2var_epi8(__m128i a, __mmask16 k, __m128i idx, __m128i b);
VPERMT2B __m128i _mm_maskz_permutex2var_epi8(__mmask16 k, __m128i a, __m128i idx, __m128i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

VPERMT2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting One Table

Opcode/ 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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	Permute single-precision floating-point values from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 7F /r VPERMT2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Permute single-precision floating-point values from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 7F /r VPERMT2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1[1]	Permute single-precision floating-point values from two tables in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EVEX.128.66.0F38.W1 7F /r VPERMT2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Permute double precision floating-point values from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 7F /r VPERMT2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Permute double precision floating-point values from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 7F /r VPERMT2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1[1]	Permute double precision floating-point values from two tables in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r,w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Permutes 16-bit/32-bit/64-bit values in the first operand and the third operand (the second source operand) using indices in the second operand (the first source operand) to select elements from the first and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same index can be reused for example for a second iteration, while the table elements being permuted are overwritten.

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.

Operation

VPERMT2W (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    id := 2
FI;
IF VL = 256
    id := 3
FI;
IF VL = 512
    id := 4
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
    i := j * 16
    off := 16*SRC1[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+15:i]=SRC1[i+id+1] ? SRC2[off+15:off]
                    : TMP_DEST[off+15:off]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMT2D/VPERMT2PS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    id := 1
FI;
IF VL = 256
    id := 2
FI;
IF VL = 512
    id := 3
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
    i := j * 32
    off := 32*SRC1[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC1[i+id+1] ? SRC2[31:0]
                   : TMP_DEST[off+31:off]
            ELSE
                DEST[i+31:i] := SRC1[i+id+1] ? SRC2[off+31:off]
                   : TMP_DEST[off+31:off]
```

```
                FI
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                                ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
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 Table 2-50, “Type E4NF Class Exception Conditions.”
VPERMT2W: See Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”

VPEXPANDB/VPEXPANDW—Expand Byte/Word Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 62 /r VPEXPANDB xmm1{k1}{z}, m128	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Expands up to 128 bits of packed byte values from m128 to xmm1 with writemask k1.
EVEX.128.66.0F38.W0 62 /r VPEXPANDB xmm1{k1}{z}, xmm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Expands up to 128 bits of packed byte values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W0 62 /r VPEXPANDB ymm1{k1}{z}, m256	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Expands up to 256 bits of packed byte values from m256 to ymm1 with writemask k1.
EVEX.256.66.0F38.W0 62 /r VPEXPANDB ymm1{k1}{z}, ymm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Expands up to 256 bits of packed byte values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W0 62 /r VPEXPANDB zmm1{k1}{z}, m512	A	V/V	AVX512_VBMI2 OR AVX10.1[1]	Expands up to 512 bits of packed byte values from m512 to zmm1 with writemask k1.
EVEX.512.66.0F38.W0 62 /r VPEXPANDB zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2 OR AVX10.1[1]	Expands up to 512 bits of packed byte values from zmm2 to zmm1 with writemask k1.
EVEX.128.66.0F38.W1 62 /r VPEXPANDW xmm1{k1}{z}, m128	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Expands up to 128 bits of packed word values from m128 to xmm1 with writemask k1.
EVEX.128.66.0F38.W1 62 /r VPEXPANDW xmm1{k1}{z}, xmm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Expands up to 128 bits of packed word values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W1 62 /r VPEXPANDW ymm1{k1}{z}, m256	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Expands up to 256 bits of packed word values from m256 to ymm1 with writemask k1.
EVEX.256.66.0F38.W1 62 /r VPEXPANDW ymm1{k1}{z}, ymm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Expands up to 256 bits of packed word values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W1 62 /r VPEXPANDW zmm1{k1}{z}, m512	A	V/V	AVX512_VBMI2 OR AVX10.1[1]	Expands up to 512 bits of packed word values from m512 to zmm1 with writemask k1.
EVEX.512.66.0F38.W1 62 /r VPEXPANDW zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2 OR AVX10.1[1]	Expands up to 512 bits of packed byte integer values from zmm2 to zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Expands (loads) up to 64 byte integer values or 32 word integer values from the source operand (memory operand) to the destination operand (register operand), based on the active elements determined by the writemask operand.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Moves 128, 256 or 512 bits of packed byte integer values from the source operand (memory operand) to the destination operand (register operand). This instruction is used to load from an int8 vector register or memory location while inserting the data into sparse elements of destination vector register using the active elements pointed out by the operand writemask.

This instruction supports memory fault suppression.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPEXPANDB

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[j] := SRC.byte[k];
        k := k + 1
        ELSE:
            IF *merging-masking*:
                *DEST.byte[j] remains unchanged*
                ELSE:                    ; zeroing-masking
                    DEST.byte[j] := 0
DEST[MAX_VL-1:VL] := 0
```

VPEXPANDW

```
(KL, VL) = (8,128), (16,256), (32, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.word[j] := SRC.word[k];
        k := k + 1
        ELSE:
            IF *merging-masking*:
                *DEST.word[j] remains unchanged*
                ELSE:                    ; zeroing-masking
                    DEST.word[j] := 0
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPEXPAND __m128i _mm_mask_expand_epi8(__m128i, __mmask16, __m128i);
VPEXPAND __m128i _mm_maskz_expand_epi8(__mmask16, __m128i);
VPEXPAND __m128i _mm_mask_expandloadu_epi8(__m128i, __mmask16, const void*);
VPEXPAND __m128i _mm_maskz_expandloadu_epi8(__mmask16, const void*);
VPEXPAND __m256i _mm256_mask_expand_epi8(__m256i, __mmask32, __m256i);

```
VPEXPAND __m256i _mm256_maskz_expand_epi8(__mmask32, __m256i);
VPEXPAND __m256i _mm256_mask_expandloadu_epi8(__m256i, __mmask32, const void*);
VPEXPAND __m256i _mm256_maskz_expandloadu_epi8(__mmask32, const void*);
VPEXPAND __m512i _mm512_mask_expand_epi8(__m512i, __mmask64, __m512i);
VPEXPAND __m512i _mm512_maskz_expand_epi8(__mmask64, __m512i);
VPEXPAND __m512i _mm512_mask_expandloadu_epi8(__m512i, __mmask64, const void*);
VPEXPAND __m512i _mm512_maskz_expandloadu_epi8(__mmask64, const void*);
VPEXPANDW __m128i _mm_mask_expand_epi16(__m128i, __mmask8, __m128i);
VPEXPANDW __m128i _mm_maskz_expand_epi16(__mmask8, __m128i);
VPEXPANDW __m128i _mm_mask_expandloadu_epi16(__m128i, __mmask8, const void*);
VPEXPANDW __m128i _mm_maskz_expandloadu_epi16(__mmask8, const void *);
VPEXPANDW __m256i _mm256_mask_expand_epi16(__m256i, __mmask16, __m256i);
VPEXPANDW __m256i _mm256_maskz_expand_epi16(__mmask16, __m256i);
VPEXPANDW __m256i _mm256_mask_expandloadu_epi16(__m256i, __mmask16, const void*);
VPEXPANDW __m256i _mm256_maskz_expandloadu_epi16(__mmask16, const void*);
VPEXPANDW __m512i _mm512_mask_expand_epi16(__m512i, __mmask32, __m512i);
VPEXPANDW __m512i _mm512_maskz_expand_epi16(__mmask32, __m512i);
VPEXPANDW __m512i _mm512_mask_expandloadu_epi16(__m512i, __mmask32, const void*);
VPEXPANDW __m512i _mm512_maskz_expandloadu_epi16(__mmask32, const void*);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”

VPEXPANDD—Load Sparse Packed Doubleword Integer Values From Dense Memory/Register

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Expand packed double-word integer values from zmm2/m512 to zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Expand (load) up to 16 contiguous doubleword integer values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPEXPANDD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] := SRC[k+31:k];
            k := k + 32
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
```

```
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPEXPANDD __m512i _mm512_mask_expandloadu_epi32(__m512i s, __mmask16 k, void * a);
VPEXPANDD __m512i _mm512_maskz_expandloadu_epi32( __mmask16 k, void * a);
VPEXPANDD __m512i _mm512_mask_expand_epi32(__m512i s, __mmask16 k, __m512i a);
VPEXPANDD __m512i _mm512_maskz_expand_epi32( __mmask16 k, __m512i a);
VPEXPANDD __m256i _mm256_mask_expandloadu_epi32(__m256i s, __mmask8 k, void * a);
VPEXPANDD __m256i _mm256_maskz_expandloadu_epi32( __mmask8 k, void * a);
VPEXPANDD __m256i _mm256_mask_expand_epi32(__m256i s, __mmask8 k, __m256i a);
VPEXPANDD __m256i _mm256_maskz_expand_epi32( __mmask8 k, __m256i a);
VPEXPANDD __m128i _mm_mask_expandloadu_epi32(__m128i s, __mmask8 k, void * a);
VPEXPANDD __m128i _mm_maskz_expandloadu_epi32( __mmask8 k, void * a);
VPEXPANDD __m128i _mm_mask_expand_epi32(__m128i s, __mmask8 k, __m128i a);
VPEXPANDD __m128i _mm_maskz_expand_epi32( __mmask8 k, __m128i a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPEXPANDQ—Load Sparse Packed Quadword Integer Values From Dense Memory/Register

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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Expand packed quad-word integer values from zmm2/m512 to zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Expand (load) up to 8 quadword integer values from the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPEXPANDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+63:i] := SRC[k+63:k];
            k := k + 64
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPEXPANDQ __m512i _mm512_mask_expandloadu_epi64(__m512i s, __mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_mask_expand_epi64(__m512i s, __mmask8 k, __m512i a);
VPEXPANDQ __m512i _mm512_maskz_expand_epi64(__mmask8 k, __m512i a);
VPEXPANDQ __m256i _mm256_mask_expandloadu_epi64(__m256i s, __mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_mask_expand_epi64(__m256i s, __mmask8 k, __m256i a);
VPEXPANDQ __m256i _mm256_maskz_expand_epi64(__mmask8 k, __m256i a);
VPEXPANDQ __m128i _mm_mask_expandloadu_epi64(__m128i s, __mmask8 k, void * a);
VPEXPANDQ __m128i _mm_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m128i _mm_mask_expand_epi64(__m128i s, __mmask8 k, __m128i a);
VPEXPANDQ __m128i _mm_maskz_expand_epi64(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

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	N/A	N/A

Description

A set of 16 or 8 doubleword/quadword memory locations pointed to by base address BASE_ADDR and index vector VINDEX with scale SCALE are gathered. The result is written into vector zmm1. The elements are specified via the VSIB (i.e., the index register is a zmm, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register (zmm1) is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.

- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same disp8*N and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

VPGATHERDD (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j]
        THEN DEST[i+31:i] := MEM[BASE_ADDR +
                    SignExtend(VINDEX[i+31:i]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+31:i] := remains unchanged*        ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

VPGATHERDQ (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j]
        THEN DEST[i+63:i] :=
            MEM[BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+63:i] := remains unchanged*        ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERDD __m512i _mm512_i32gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERDD __m512i _mm512_mask_i32gather_epi32(__m512i s, __mmask16 k, __m512i vdx, void * base, int scale);
VPGATHERDD __m256i _mm256_mmask_i32gather_epi32(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDD __m128i _mm_mmask_i32gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m512i _mm512_i32logather_epi64(__m256i vdx, void * base, int scale);
VPGATHERDQ __m512i _mm512_mask_i32logather_epi64(__m512i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDQ __m256i _mm256_mmask_i32logather_epi64(__m256i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m128i _mm_mmask_i32gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-61, “Type E12 Class Exception Conditions.”

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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

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	N/A	N/A

Description

A set of 8 doubleword/quadword memory locations pointed to by base address BASE_ADDR and index vector VINDEX with scale SCALE are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same disp8*N and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement
```

VPGATHERQD (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j]
        THEN DEST[i+31:i] := MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+31:i] := remains unchanged*        ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL/2] := 0
```

VPGATHERQQ (EVEX encoded version)

```
(KL, VL) = (2, 64), (4, 128), (8, 256)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j]
        THEN DEST[i+63:i] :=
            MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+63:i] := remains unchanged*        ; Only merging masking is allowed
    FI;
```

```
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPGATHERQD __m256i _mm512_i64gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERQD __m256i _mm512_mask_i64gather_epi32lo(__m256i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQD __m128i _mm256_mask_i64gather_epi32lo(__m128i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQD __m128i _mm_mask_i64gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERQQ __m512i _mm512_i64gather_epi64( __m512i vdx, void * base, int scale);
VPGATHERQQ __m512i _mm512_mask_i64gather_epi64(__m512i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQQ __m256i _mm256_mask_i64gather_epi64(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQQ __m128i _mm_mask_i64gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-61, “Type E12 Class Exception Conditions.”

VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values

Opcode/ 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 AND AVX512CD) OR AVX10.1[1]	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 AND AVX512CD) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512CD) OR AVX10.1[1]	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 AND AVX512CD) OR AVX10.1[1]	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 OR AVX10.1[1]	Count the number of leading zero bits in each qword element of zmm2/m512/m64bcst using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Counts the number of leading most significant zero bits in each dword or qword element of the source operand (the second operand) and stores the results in the destination register (the first operand) according to the writemask. If an element is zero, the result for that element is the operand size of the element.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPLZCNTD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j*32
    IF MaskBit(j) OR *no writemask*
         THEN
              temp := 32
              DEST[i+31:i] := 0
              WHILE (temp > 0) AND (SRC[i+temp-1] = 0)
              DO
                   temp := temp – 1
                   DEST[i+31:i] := DEST[i+31:i] + 1
              OD
         ELSE
           IF *merging-masking*
              THEN *DEST[i+31:i] remains unchanged*
              ELSE DEST[i+31:i] := 0
           FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPLZCNTQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j*64
    IF MaskBit(j) OR *no writemask*
         THEN
              temp := 64
              DEST[i+63:i] := 0
              WHILE (temp > 0) AND (SRC[i+temp-1] = 0)
              DO
                   temp := temp – 1
                   DEST[i+63:i] := DEST[i+63:i] + 1
              OD
         ELSE
           IF *merging-masking*
              THEN *DEST[i+63:i] remains unchanged*
              ELSE DEST[i+63:i] := 0
           FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPLZCNTD __m512i _mm512_lzcnt_epi32(__m512i a);
VPLZCNTD __m512i _mm512_mask_lzcnt_epi32(__m512i s, __mmask16 m, __m512i a);
VPLZCNTD __m512i _mm512_maskz_lzcnt_epi32(__mmask16 m, __m512i a);
VPLZCNTQ __m512i _mm512_lzcnt_epi64(__m512i a);
VPLZCNTQ __m512i _mm512_mask_lzcnt_epi64(__m512i s, __mmask8 m, __m512i a);
VPLZCNTQ __m512i _mm512_maskz_lzcnt_epi64(__mmask8 m, __m512i a);
VPLZCNTD __m256i _mm256_lzcnt_epi32(__m256i a);
VPLZCNTD __m256i _mm256_mask_lzcnt_epi32(__m256i s, __mmask8 m, __m256i a);
VPLZCNTD __m256i _mm256_maskz_lzcnt_epi32(__mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_lzcnt_epi64(__m256i a);
VPLZCNTQ __m256i _mm256_mask_lzcnt_epi64(__m256i s, __mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_maskz_lzcnt_epi64(__mmask8 m, __m256i a);
VPLZCNTD __m128i _mm_lzcnt_epi32(__m128i a);
VPLZCNTD __m128i _mm_mask_lzcnt_epi32(__m128i s, __mmask8 m, __m128i a);
VPLZCNTD __m128i _mm_maskz_lzcnt_epi32(__mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_lzcnt_epi64(__m128i a);
VPLZCNTQ __m128i _mm_mask_lzcnt_epi64(__m128i s, __mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_maskz_lzcnt_epi64(__mmask8 m, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

VPMADD52HUQ—Packed Multiply of Unsigned 52-Bit Unsigned Integers and Add High 52-Bit Products to 64-Bit Accumulators

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 AND AVX512VL) OR AVX10.1[1]	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 AND AVX512VL) OR AVX10.1[1]	Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the high 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 B5 /r VPMADD52HUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	A	V/V	AVX512_IFMA OR AVX10.1[1]	Multiply unsigned 52-bit integers in zmm2 and zmm3/m512 and add the high 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m(r)	N/A

Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The high 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.

Operation

VPMADD52HUQ (EVEX encoded)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64;
    IF k1[j] OR *no writemask* THEN
        IF src2 is Memory AND EVEX.b=1 THEN
            tsrc2[63:0] := ZeroExtend64(src2[51:0]);
        ELSE
            tsrc2[63:0] := ZeroExtend64(src2[i+51:i];
        FI;
        Temp128[127:0] := ZeroExtend64(src1[i+51:i]) * tsrc2[63:0];
        Temp2[63:0] := DEST[i+63:i] + ZeroExtend64(temp128[103:52]) ;
        DEST[i+63:i] := Temp2[63:0];
    ELSE
        IF *zeroing-masking* THEN
            DEST[i+63:i] := 0;
        ELSE *merge-masking*
            DEST[i+63:i] is unchanged;
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMADD52HUQ __m512i _mm512_madd52hi_epu64( __m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m512i _mm512_mask_madd52hi_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m512i _mm512_maskz_madd52hi_epu64( __mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m256i _mm256_madd52hi_epu64( __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m256i _mm256_mask_madd52hi_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m256i _mm256_maskz_madd52hi_epu64( __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m128i _mm_madd52hi_epu64( __m128i a, __m128i b, __m128i c);
VPMADD52HUQ __m128i _mm_mask_madd52hi_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b, __m128i c);
VPMADD52HUQ __m128i _mm_maskz_madd52hi_epu64( __mmask8 k, __m128i a, __m128i b, __m128i c);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”

VPMADD52LUQ—Packed Multiply of Unsigned 52-Bit Integers and Add the Low 52-Bit Products to Qword Accumulators

Opcode/ 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 AND AVX512VL) OR AVX10.1[1]	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 AND AVX512VL) OR AVX10.1[1]	Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the low 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 B4 /r VPMADD52LUQ zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst	A	V/V	AVX512_IFMA OR AVX10.1[1]	Multiply unsigned 52-bit integers in zmm2 and zmm3/m512 and add the low 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The low 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.

Operation

VPMADD52LUQ (EVEX encoded)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64;
    IF k1[j] OR *no writemask* THEN
        IF src2 is Memory AND EVEX.b=1 THEN
            tsrc2[63:0] := ZeroExtend64(src2[51:0]);
        ELSE
            tsrc2[63:0] := ZeroExtend64(src2[i+51:i];
        FI;
        Temp128[127:0] := ZeroExtend64(src1[i+51:i]) * tsrc2[63:0];
        Temp2[63:0] := DEST[i+63:i] + ZeroExtend64(temp128[51:0]) ;
        DEST[i+63:i] := Temp2[63:0];
    ELSE
        IF *zeroing-masking* THEN
            DEST[i+63:i] := 0;
        ELSE *merge-masking*
            DEST[i+63:i] is unchanged;
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMADD52LUQ __m512i _mm512_madd52lo_epu64(__m512i a, __m512i b, __m512i c);
VPMADD52LUQ __m512i _mm512_mask_madd52lo_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52LUQ __m512i _mm512_maskz_madd52lo_epu64(__mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52LUQ __m256i _mm256_madd52lo_epu64(__m256i a, __m256i b, __m256i c);
VPMADD52LUQ __m256i _mm256_mask_madd52lo_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52LUQ __m256i _mm256_maskz_madd52lo_epu64(__mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52LUQ __m128i _mm_madd52lo_epu64(__m128i a, __m128i b, __m128i c);
VPMADD52LUQ __m128i _mm_mask_madd52lo_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b, __m128i c);
VPMADD52LUQ __m128i _mm_maskz_madd52lo_epu64(__mmask8 k, __m128i a, __m128i b, __m128i c);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, "Type E4 Class Exception Conditions."

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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in ZMM1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a vector register to a mask register. Each element in the destination register is set to 1 or 0 depending on the value of most significant bit of the corresponding element in the source register.

The source operand is a ZMM/YMM/XMM register. The destination operand is a mask register.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVB2M (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF SRC[i+7]
        THEN    DEST[j] := 1
        ELSE    DEST[j] := 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPMOVW2M (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF SRC[i+15]
        THEN    DEST[j] := 1
        ELSE    DEST[j] := 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPMOVD2M (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF SRC[i+31]
        THEN    DEST[j] := 1
        ELSE    DEST[j] := 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPMOVQ2M (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF SRC[i+63]
        THEN    DEST[j] := 1
        ELSE    DEST[j] := 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVB2M __mmask64 _mm512_movepi8_mask(__m512i);
VPMOVD2M __mmask16 _mm512_movepi32_mask(__m512i);
VPMOVQ2M __mmask8 _mm512_movepi64_mask(__m512i);
VPMOVW2M __mmask32 _mm512_movepi16_mask(__m512i);
VPMOVB2M __mmask32 _mm256_movepi8_mask(__m256i);
VPMOVD2M __mmask8 _mm256_movepi32_mask(__m256i);
VPMOVQ2M __mmask8 _mm256_movepi64_mask(__m256i);
VPMOVW2M __mmask16 _mm256_movepi16_mask(__m256i);
VPMOVB2M __mmask16 _mm_movepi8_mask(__m128i);
VPMOVD2M __mmask8 _mm_movepi32_mask(__m128i);
VPMOVQ2M __mmask8 _mm_movepi64_mask(__m128i);
VPMOVW2M __mmask8 _mm_movepi16_mask(__m128i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-55, “Type E7NM Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVDB/VPMOVSDB/VPMOVUSDB—Down Convert DWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 31 /r VPMOVDB xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 4 packed double-word integers from xmm2 into 4 packed byte integers in xmm1/m32 with truncation under writemask k1.
EVEX.128.F3.0F38.W0 21 /r VPMOVSDB xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 4 packed signed double-word integers from xmm2 into 4 packed signed byte integers in xmm1/m32 using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 4 packed unsigned double-word integers from xmm2 into 4 packed unsigned byte integers in xmm1/m32 using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 31 /r VPMOVDB xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 8 packed double-word integers from ymm2 into 8 packed byte integers in xmm1/m64 with truncation under writemask k1.
EVEX.256.F3.0F38.W0 21 /r VPMOVSDB xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 8 packed signed double-word integers from ymm2 into 8 packed signed byte integers in xmm1/m64 using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 8 packed unsigned double-word integers from ymm2 into 8 packed unsigned byte integers in xmm1/m64 using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 31 /r VPMOVDB xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Converts 16 packed double-word integers from zmm2 into 16 packed byte integers in xmm1/m128 with truncation under writemask k1.
EVEX.512.F3.0F38.W0 21 /r VPMOVSDB xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Converts 16 packed signed double-word integers from zmm2 into 16 packed signed byte integers in xmm1/m128 using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Converts 16 packed unsigned double-word integers from zmm2 into 16 packed unsigned byte integers in xmm1/m128 using unsigned saturation under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVDB down converts 32-bit integer elements in the source operand (the second operand) into packed bytes using truncation. VPMOVSDB converts signed 32-bit integers into packed signed bytes using signed saturation. VPMOVUSDB 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

VPMOVDB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateDoubleWordToByte (SRC[m+31:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;
```

VPMOVDB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateDoubleWordToByte (SRC[m+31:m])
        ELSE *DEST[i+7:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR
```

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

VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB 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);
VPMOVDB __m128i _mm256_cvtepi32_epi8(__m256i a);
VPMOVDB __m128i _mm256_mask_cvtepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVDB __m128i _mm256_maskz_cvtepi32_epi8(__mmask8 k, __m256i b);
VPMOVDB void _mm256_mask_cvtepi32_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVDB __m128i _mm_cvtepi32_epi8(__m128i a);
VPMOVDB __m128i _mm_mask_cvtepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVDB __m128i _mm_maskz_cvtepi32_epi8(__mmask8 k, __m128i b);
VPMOVDB void _mm_mask_cvtepi32_storeu_epi8(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-53, “Type E6 Class Exception Conditions.”

Additionally:

#UD If EVEX.vvvv != 1111B.

VPMOVDW/VPMOVSDW/VPMOVUSDW—Down Convert DWord to Word

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 33 /r VPMOVDW xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 4 packed double-word integers from xmm2 into 4 packed word integers in xmm1/m64 with truncation under writemask k1.
EVEX.128.F3.0F38.W0 23 /r VPMOVSDW xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 4 packed signed double-word integers from xmm2 into 4 packed signed word integers in ymm1/m64 using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 13 /r VPMOVUSDW xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 4 packed unsigned double-word integers from xmm2 into 4 packed unsigned word integers in xmm1/m64 using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 33 /r VPMOVDW xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 8 packed double-word integers from ymm2 into 8 packed word integers in xmm1/m128 with truncation under writemask k1.
EVEX.256.F3.0F38.W0 23 /r VPMOVSDW xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 8 packed signed double-word integers from ymm2 into 8 packed signed word integers in xmm1/m128 using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 13 /r VPMOVUSDW xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 8 packed unsigned double-word integers from ymm2 into 8 packed unsigned word integers in xmm1/m128 using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 33 /r VPMOVDW ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Converts 16 packed double-word integers from zmm2 into 16 packed word integers in ymm1/m256 with truncation under writemask k1.
EVEX.512.F3.0F38.W0 23 /r VPMOVSDW ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Converts 16 packed signed double-word integers from zmm2 into 16 packed signed word integers in ymm1/m256 using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 13 /r VPMOVUSDW ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Converts 16 packed unsigned double-word integers from zmm2 into 16 packed unsigned word integers in ymm1/m256 using unsigned saturation under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVDW down converts 32-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSDW converts signed 32-bit integers into packed signed words using signed saturation. VPMOVUSDW convert unsigned double-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVDW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TruncateDoubleWordToWord (SRC[m+31:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVDW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TruncateDoubleWordToWord (SRC[m+31:m])
        ELSE
            *DEST[i+15:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR
```

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 Table 2-53, “Type E6 Class Exception Conditions.”

Additionally:

#UD If EVEX.vvvv != 1111B.

VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 28 /r VPMOVM2B xmm1, k1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Sets each byte in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W0 28 /r VPMOVM2B ymm1, k1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Sets each byte in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W0 28 /r VPMOVM2B zmm1, k1	RM	V/V	AVX512BW OR AVX10.1[1]	Sets each byte in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W1 28 /r VPMOVM2W xmm1, k1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Sets each word in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W1 28 /r VPMOVM2W ymm1, k1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Sets each word in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W1 28 /r VPMOVM2W zmm1, k1	RM	V/V	AVX512BW OR AVX10.1[1]	Sets each word in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W0 38 /r VPMOVM2D xmm1, k1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Sets each doubleword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W0 38 /r VPMOVM2D ymm1, k1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Sets each doubleword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W0 38 /r VPMOVM2D zmm1, k1	RM	V/V	AVX512DQ OR AVX10.1[1]	Sets each doubleword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W1 38 /r VPMOVM2Q xmm1, k1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Sets each quadword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W1 38 /r VPMOVM2Q ymm1, k1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1[1]	Sets each quadword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W1 38 /r VPMOVM2Q zmm1, k1	RM	V/V	AVX512DQ OR AVX10.1[1]	Sets each quadword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a mask register to a vector register. Each element in the destination register is set to all 1's or all 0's depending on the value of the corresponding bit in the source mask register.

The source operand is a mask register. The destination operand is a ZMM/YMM/XMM register.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVM2B (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF SRC[j]
        THEN    DEST[i+7:i] := -1
        ELSE    DEST[i+7:i] := 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVM2W (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF SRC[j]
        THEN    DEST[i+15:i] := -1
        ELSE    DEST[i+15:i] := 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVM2D (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF SRC[j]
        THEN    DEST[i+31:i] := -1
        ELSE    DEST[i+31:i] := 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVM2Q (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF SRC[j]
        THEN    DEST[i+63:i] := -1
        ELSE    DEST[i+63:i] := 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVM2B __m512i _mm512_movm_epi8(__mmask64);
VPMOVM2D __m512i _mm512_movm_epi32(__mmask8);
VPMOVM2Q __m512i _mm512_movm_epi64(__mmask16);
VPMOVM2W __m512i _mm512_movm_epi16(__mmask32);
VPMOVM2B __m256i _mm256_movm_epi8(__mmask32);
VPMOVM2D __m256i _mm256_movm_epi32(__mmask8);
VPMOVM2Q __m256i _mm256_movm_epi64(__mmask8);
VPMOVM2W __m256i _mm256_movm_epi16(__mmask16);
VPMOVM2B __m128i _mm_movm_epi8(__mmask16);
VPMOVM2D __m128i _mm_movm_epi32(__mmask8);
VPMOVM2Q __m128i _mm_movm_epi64(__mmask8);
VPMOVM2W __m128i _mm_movm_epi16(__mmask8);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-55, “Type E7NM Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVQB/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 32 /r VPMOVQB xmm1/m16 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 2 packed quad-word integers from xmm2 into 2 packed byte integers in xmm1/m16 with truncation under writemask k1.
EVEX.128.F3.0F38.W0 22 /r VPMOVSQB xmm1/m16 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 2 packed signed quad-word integers from xmm2 into 2 packed signed byte integers in xmm1/m16 using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m16 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned byte integers in xmm1/m16 using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 32 /r VPMOVQB xmm1/m32 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 4 packed quad-word integers from ymm2 into 4 packed byte integers in xmm1/m32 with truncation under writemask k1.
EVEX.256.F3.0F38.W0 22 /r VPMOVSQB xmm1/m32 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed byte integers in xmm1/m32 using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m32 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned byte integers in xmm1/m32 using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 32 /r VPMOVQB xmm1/m64 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Converts 8 packed quad-word integers from zmm2 into 8 packed byte integers in xmm1/m64 with truncation under writemask k1.
EVEX.512.F3.0F38.W0 22 /r VPMOVSQB xmm1/m64 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed byte integers in xmm1/m64 using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m64 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned byte integers in xmm1/m64 using unsigned saturation under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Eighth Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVQB down converts 64-bit integer elements in the source operand (the second operand) into packed byte elements using truncation. VPMOVSQB converts signed 64-bit integers into packed signed bytes using signed saturation. VPMOVUSQB convert unsigned quad-word values into unsigned byte values using unsigned saturation. The source operand is a vector register. The destination operand is an XMM register or a memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:64) of the destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVQB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateQuadWordToByte (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/8] := 0;
```

VPMOVQB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateQuadWordToByte (SRC[m+63:m])
        ELSE
            *DEST[i+7:i] remains unchanged*         ; merging-masking
    FI;
ENDFOR
```

VPMOVSQB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateSignedQuadWordToByte (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
```

```
        FI;
    ENDFOR
    DEST[MAXVL-1:VL/8] := 0;
```

VPMOVSQB instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 8
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+7:i] := SaturateSignedQuadWordToByte (SRC[m+63:m])
            ELSE
                *DEST[i+7:i] remains unchanged*        ; merging-masking
        FI;
    ENDFOR
```

VPMOVUSQB instruction (EVEX encoded versions) when dest is a register

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 8
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+7:i] := SaturateUnsignedQuadWordToByte (SRC[m+63:m])
            ELSE
                IF *merging-masking*                   ; merging-masking
                    THEN *DEST[i+7:i] remains unchanged*
                    ELSE *zeroing-masking*             ; zeroing-masking
                        DEST[i+7:i] := 0
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL/8] := 0;
```

VPMOVUSQB instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 8
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+7:i] := SaturateUnsignedQuadWordToByte (SRC[m+63:m])
            ELSE
                *DEST[i+7:i] remains unchanged*        ; merging-masking
        FI;
    ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

```
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 Table 2-53, “Type E6 Class Exception Conditions.”

Additionally:

#UD If EVEX.vvvv != 1111B.

VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 35 /r VPMOVQD xmm1/m128 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 2 packed quad-word integers from xmm2 into 2 packed double-word integers in xmm1/m128 with truncation subject to writemask k1.
EVEX.128.F3.0F38.W0 25 /r VPMOVSQD xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 2 packed signed quad-word integers from xmm2 into 2 packed signed double-word integers in xmm1/m64 using signed saturation subject to writemask k1.
EVEX.128.F3.0F38.W0 15 /r VPMOVUSQD xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned double-word integers in xmm1/m64 using unsigned saturation subject to writemask k1.
EVEX.256.F3.0F38.W0 35 /r VPMOVQD xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 4 packed quad-word integers from ymm2 into 4 packed double-word integers in xmm1/m128 with truncation subject to writemask k1.
EVEX.256.F3.0F38.W0 25 /r VPMOVSQD xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed double-word integers in xmm1/m128 using signed saturation subject to writemask k1.
EVEX.256.F3.0F38.W0 15 /r VPMOVUSQD xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned double-word integers in xmm1/m128 using unsigned saturation subject to writemask k1.
EVEX.512.F3.0F38.W0 35 /r VPMOVQD ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Converts 8 packed quad-word integers from zmm2 into 8 packed double-word integers in ymm1/m256 with truncation subject to writemask k1.
EVEX.512.F3.0F38.W0 25 /r VPMOVSQD ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed double-word integers in ymm1/m256 using signed saturation subject to writemask k1.
EVEX.512.F3.0F38.W0 15 /r VPMOVUSQD ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned double-word integers in ymm1/m256 using unsigned saturation subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed double-words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed doublewords using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned double-word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted doubleword elements are written to the destination operand (the first operand) from the least-significant doubleword. Doubleword elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TruncateQuadWordToDWord (SRC[m+63:m])
        ELSE *zeroing-masking*                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVQD instruction (EVEX encoded version) memory form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TruncateQuadWordToDWord (SRC[m+63:m])
        ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR
```

VPMOVSQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SaturateSignedQuadWordToDWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVSQD instruction (EVEX encoded version) memory form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SaturateSignedQuadWordToDWord (SRC[m+63:m])
        ELSE *DEST[i+31:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

VPMOVUSQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                          ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                    ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVUSQD instruction (EVEX encoded version) memory form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
        ELSE *DEST[i+31:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPMOVQD __m256i _mm512_cvtepi64_epi32( __m512i a);
VPMOVQD __m256i _mm512_mask_cvtepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVQD __m256i _mm512_maskz_cvtepi64_epi32( __mmask8 k, __m512i a);
VPMOVQD void _mm512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVSQD __m256i _mm512_cvtsepi64_epi32( __m512i a);
VPMOVSQD __m256i _mm512_mask_cvtsepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVSQD __m256i _mm512_maskz_cvtsepi64_epi32( __mmask8 k, __m512i a);
VPMOVSQD void _mm512_mask_cvtsepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m256i _mm512_cvtusepi64_epi32( __m512i a);
VPMOVUSQD __m256i _mm512_mask_cvtusepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVUSQD __m256i _mm512_maskz_cvtusepi64_epi32( __mmask8 k, __m512i a);
VPMOVUSQD void _mm512_mask_cvtusepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m128i _mm256_cvtusepi64_epi32(__m256i a);
VPMOVUSQD __m128i _mm256_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm256_maskz_cvtusepi64_epi32( __mmask8 k, __m256i b);
VPMOVUSQD void _mm256_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm_cvtusepi64_epi32(__m128i a);
VPMOVUSQD __m128i _mm_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQD __m128i _mm_maskz_cvtusepi64_epi32( __mmask8 k, __m128i b);
VPMOVUSQD void _mm_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm256_cvtsepi64_epi32(__m256i a);
VPMOVSQD __m128i _mm256_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm256_maskz_cvtsepi64_epi32( __mmask8 k, __m256i b);
VPMOVSQD void _mm256_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm_cvtsepi64_epi32(__m128i a);
VPMOVSQD __m128i _mm_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm_maskz_cvtsepi64_epi32( __mmask8 k, __m128i b);
VPMOVSQD void _mm_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVQD __m128i _mm256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i _mm256_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVQD __m128i _mm256_maskz_cvtepi64_epi32( __mmask8 k, __m256i b);
VPMOVQD void _mm256_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVQD __m128i _mm_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i _mm_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i _mm_maskz_cvtepi64_epi32( __mmask8 k, __m128i b);
VPMOVQD void _mm_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-53, “Type E6 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 34 /r VPMOVQW xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 2 packed quad-word integers from xmm2 into 2 packed word integers in xmm1/m32 with truncation under writemask k1.
EVEX.128.F3.0F38.W0 24 /r VPMOVSQW xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed word integers in xmm1/m32 using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 14 /r VPMOVUSQW xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned word integers in xmm1/m32 using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 34 /r VPMOVQW xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 4 packed quad-word integers from ymm2 into 4 packed word integers in xmm1/m64 with truncation under writemask k1.
EVEX.256.F3.0F38.W0 24 /r VPMOVSQW xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed word integers in xmm1/m64 using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 14 /r VPMOVUSQW xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned word integers in xmm1/m64 using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 34 /r VPMOVQW xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Converts 8 packed quad-word integers from zmm2 into 8 packed word integers in xmm1/m128 with truncation under writemask k1.
EVEX.512.F3.0F38.W0 24 /r VPMOVSQW xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed word integers in xmm1/m128 using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 14 /r VPMOVUSQW xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1[1]	Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned word integers in xmm1/m128 using unsigned saturation under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed words using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:128/64/32) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVQW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TruncateQuadWordToWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;
```

VPMOVQW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TruncateQuadWordToWord (SRC[m+63:m])
        ELSE
            *DEST[i+15:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR
```

VPMOVSQW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateSignedQuadWordToWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;
```

VPMOVSQW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateSignedQuadWordToWord (SRC[m+63:m])
        ELSE
            *DEST[i+15:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR
```

VPMOVUSQW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateUnsignedQuadWordToWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;
```

VPMOVUSQW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateUnsignedQuadWordToWord (SRC[m+63:m])
        ELSE
            *DEST[i+15:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQW __m128i _mm512_cvtepi64_epi16(__m512i a);
VPMOVQW __m128i _mm512_mask_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_maskz_cvtepi64_epi16(__mmask8 k, __m512i a);
VPMOVQW void _mm512_mask_cvtepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVSQW __m128i _mm512_cvtsepi64_epi16(__m512i a);
VPMOVSQW __m128i _mm512_mask_cvtsepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVSQW __m128i _mm512_maskz_cvtsepi64_epi16(__mmask8 k, __m512i a);
VPMOVSQW void _mm512_mask_cvtsepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_cvtusepi64_epi16(__m512i a);
VPMOVUSQW __m128i _mm512_mask_cvtusepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_maskz_cvtusepi64_epi16(__mmask8 k, __m512i a);
VPMOVUSQW void _mm512_mask_cvtusepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m128i _mm256_cvtusepi64_epi32(__m256i a);
VPMOVUSQD __m128i _mm256_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm256_maskz_cvtusepi64_epi32(__mmask8 k, __m256i b);
VPMOVUSQD void _mm256_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm_cvtusepi64_epi32(__m128i a);
VPMOVUSQD __m128i _mm_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQD __m128i _mm_maskz_cvtusepi64_epi32(__mmask8 k, __m128i b);
VPMOVUSQD void _mm_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm256_cvtsepi64_epi32(__m256i a);
VPMOVSQD __m128i _mm256_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm256_maskz_cvtsepi64_epi32(__mmask8 k, __m256i b);
VPMOVSQD void _mm256_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm_cvtsepi64_epi32(__m128i a);
VPMOVSQD __m128i _mm_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm_maskz_cvtsepi64_epi32(__mmask8 k, __m128i b);
VPMOVSQD void _mm_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVQD __m128i _mm256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i _mm256_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVQD __m128i _mm256_maskz_cvtepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void _mm256_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVQD __m128i _mm_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i _mm_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i _mm_maskz_cvtepi64_epi32(__mmask8 k, __m128i b);
VPMOVQD void _mm_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-53, “Type E6 Class Exception Conditions.”

Additionally:

#UD If EVEX.vvvv != 1111B.

VPMOVWB/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 VPMOVWB xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Converts 8 packed word integers from xmm2 into 8 packed bytes in xmm1/m64 with truncation under writemask k1.
EVEX.128.F3.0F38.W0 20 /r VPMOVSWB xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Converts 8 packed signed word integers from xmm2 into 8 packed signed bytes in xmm1/m64 using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 10 /r VPMOVUSWB xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Converts 8 packed unsigned word integers from xmm2 into 8 packed unsigned bytes in 8mm1/m64 using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 30 /r VPMOVWB xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Converts 16 packed word integers from ymm2 into 16 packed bytes in xmm1/m128 with truncation under writemask k1.
EVEX.256.F3.0F38.W0 20 /r VPMOVSWB xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Converts 16 packed signed word integers from ymm2 into 16 packed signed bytes in xmm1/m128 using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 10 /r VPMOVUSWB xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1[1]	Converts 16 packed unsigned word integers from ymm2 into 16 packed unsigned bytes in xmm1/m128 using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 30 /r VPMOVWB ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512BW OR AVX10.1[1]	Converts 32 packed word integers from zmm2 into 32 packed bytes in ymm1/m256 with truncation under writemask k1.
EVEX.512.F3.0F38.W0 20 /r VPMOVSWB ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512BW OR AVX10.1[1]	Converts 32 packed signed word integers from zmm2 into 32 packed signed bytes in ymm1/m256 using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 10 /r VPMOVUSWB ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512BW OR AVX10.1[1]	Converts 32 packed unsigned word integers from zmm2 into 32 packed unsigned bytes in ymm1/m256 using unsigned saturation under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVWB down converts 16-bit integers into packed bytes using truncation. VPMOVSWB converts signed 16-bit integers into packed signed bytes using signed saturation. VPMOVUSWB convert unsigned word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVWB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO Kl-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateWordToByte (SRC[m+15:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVWB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO Kl-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateWordToByte (SRC[m+15:m])
        ELSE
            *DEST[i+7:i] remains unchanged*         ; merging-masking
    FI;
ENDFOR
```

VPMOVSWB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO Kl-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateSignedWordToByte (SRC[m+15:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVSWB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO Kl-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateSignedWordToByte (SRC[m+15:m])
        ELSE
            *DEST[i+7:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

VPMOVUSWB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO Kl-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateUnsignedWordToByte (SRC[m+15:m])
        ELSE
            IF *merging-masking*                   ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*             ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVUSWB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO Kl-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateUnsignedWordToByte (SRC[m+15:m])
        ELSE
            *DEST[i+7:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

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_cvtsepi16_epi8(__m128i a);
VPMOVSWB __m128i _mm_mask_cvtsepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSWB __m128i _mm_maskz_cvtsepi16_epi8(__mmask8 k, __m128i b);
VPMOVSWB void _mm_mask_cvtsepi16_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVWB __m128i _mm256_cvtepi16_epi8(__m256i a);
VPMOVWB __m128i _mm256_mask_cvtepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVWB __m128i _mm256_maskz_cvtepi16_epi8(__mmask16 k, __m256i b);
VPMOVWB void _mm256_mask_cvtepi16_storeu_epi8(void * , __mmask16 k, __m256i b);
VPMOVWB __m128i _mm_cvtepi16_epi8(__m128i a);
VPMOVWB __m128i _mm_mask_cvtepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVWB __m128i _mm_maskz_cvtepi16_epi8(__mmask8 k, __m128i b);
VPMOVWB void _mm_mask_cvtepi16_storeu_epi8(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-53, “Type E6 Class Exception Conditions.”

Additionally:

#UD If EVEX.vvvv != 1111B.

VPMULTISHIFTQB—Select Packed Unaligned Bytes From Quadword Sources

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 83 /r VPMULTISHIFTQB xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst	A	V/V	(AVX512VL AND AVX512_VBMI) OR AVX10.1[1]	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	(AVX512VL AVX512_VBMI) OR AVX10.1[1]	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 OR AVX10.1[1]	Select unaligned bytes from qwords in zmm3/m512/m64bcst using control bytes in zmm2, write byte results to zmm1 under k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction selects eight unaligned bytes from each input qword element of the second source operand (the third operand) and writes eight assembled bytes for each qword element in the destination operand (the first operand). Each byte result is selected using a byte-granular shift control within the corresponding qword element of the first source operand (the second operand). Each byte result in the destination operand is updated under the writemask k1.

Only the low 6 bits of each control byte are used to select an 8-bit slot to extract the output byte from the qword data in the second source operand. The starting bit of the 8-bit slot can be unaligned relative to any byte boundary and is extracted from the input qword source at the location specified in the low 6-bit of the control byte. If the 8-bit slot would exceed the qword boundary, the out-of-bound portion of the 8-bit slot is wrapped back to start from bit 0 of the input qword element.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register.

Operation

```
VPMULTISHIFTQB DEST, SRC1, SRC2 (EVEX encoded version)
(KL, VL) = (2, 128),(4, 256), (8, 512)
FOR i := 0 TO KL-1
    IF EVEX.b=1 AND src2 is memory THEN
            tcur := src2.qword[0]; //broadcasting
    ELSE
            tcur := src2.qword[i];
    FI;
    FOR j := 0 to 7
        ctrl := src1.qword[i].byte[j] & 63;
        FOR k := 0 to 7
            res.bit[k] := tcur.bit[ (ctrl+k) mod 64 ];
        ENDFOR
        IF k1[i*8+j] or no writemask THEN
            DEST.qword[i].byte[j] := res;
        ELSE IF zeroing-masking THEN
            DEST.qword[i].byte[j] := 0;
    ENDFOR
ENDFOR
DEST.qword[MAX_VL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULTISHIFTQB __m512i _mm512_multishift_epi64_epi8( __m512i a, __m512i b);
VPMULTISHIFTQB __m512i _mm512_mask_multishift_epi64_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMULTISHIFTQB __m512i _mm512_maskz_multishift_epi64_epi8( __mmask64 k, __m512i a, __m512i b);
VPMULTISHIFTQB __m256i _mm256_multishift_epi64_epi8( __m256i a, __m256i b);
VPMULTISHIFTQB __m256i _mm256_mask_multishift_epi64_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMULTISHIFTQB __m256i _mm256_maskz_multishift_epi64_epi8( __mmask32 k, __m256i a, __m256i b);
VPMULTISHIFTQB __m128i _mm_multishift_epi64_epi8( __m128i a, __m128i b);
VPMULTISHIFTQB __m128i _mm_mask_multishift_epi64_epi8(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULTISHIFTQB __m128i _mm_maskz_multishift_epi64_epi8( __mmask8 k, __m128i a, __m128i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-50, “Type E4NF Class Exception Conditions.”

VPOPCNT—Return the Count of Number of Bits Set to 1 in BYTE/WORD/DWORD/QWORD

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 54 /r VPOPCNTB xmm1{k1}{z}, xmm2/m128	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1[1]	Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W0 54 /r VPOPCNTB ymm1{k1}{z}, ymm2/m256	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1[1]	Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W0 54 /r VPOPCNTB zmm1{k1}{z}, zmm2/m512	A	V/V	AVX512_BITALG OR AVX10.1[1]	Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.
EVEX.128.66.0F38.W1 54 /r VPOPCNTW xmm1{k1}{z}, xmm2/m128	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1[1]	Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W1 54 /r VPOPCNTW ymm1{k1}{z}, ymm2/m256	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1[1]	Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W1 54 /r VPOPCNTW zmm1{k1}{z}, zmm2/m512	A	V/V	AVX512_BITALG OR AVX10.1[1]	Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.
EVEX.128.66.0F38.W0 55 /r VPOPCNTD xmm1{k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512_VPOPCNTDQ AND AVX512VL) OR AVX10.1[1]	Counts the number of bits set to one in xmm2/m128/m32bcst and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W0 55 /r VPOPCNTD ymm1{k1}{z}, ymm2/m256/m32bcst	B	V/V	(AVX512_VPOPCNTDQ AND AVX512VL) OR AVX10.1[1]	Counts the number of bits set to one in ymm2/m256/m32bcst and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W0 55 /r VPOPCNTD zmm1{k1}{z}, zmm2/m512/m32bcst	B	V/V	AVX512_VPOPCNTDQ OR AVX10.1[1]	Counts the number of bits set to one in zmm2/m512/m32bcst and puts the result in zmm1 with writemask k1.
EVEX.128.66.0F38.W1 55 /r VPOPCNTQ xmm1{k1}{z}, xmm2/m128/m64bcst	B	V/V	(AVX512_VPOPCNTDQ AND AVX512VL) OR AVX10.1[1]	Counts the number of bits set to one in xmm2/m128/m32bcst and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W1 55 /r VPOPCNTQ ymm1{k1}{z}, ymm2/m256/m64bcst	B	V/V	(AVX512_VPOPCNTDQ AND AVX512VL) OR AVX10.1[1]	Counts the number of bits set to one in ymm2/m256/m32bcst and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W1 55 /r VPOPCNTQ zmm1{k1}{z}, zmm2/m512/m64bcst	B	V/V	AVX512_VPOPCNTDQ OR AVX10.1[1]	Counts the number of bits set to one in zmm2/m512/m64bcst and puts the result in zmm1 with writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction counts the number of bits set to one in each byte, word, dword or qword element of its source (e.g., zmm2 or memory) and places the results in the destination register (zmm1). This instruction supports memory fault suppression.

Operation

VPOPCNTB

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.byte[j] := POPCNT(SRC.byte[j])
    ELSE IF *merging-masking*:
        *DEST.byte[j] remains unchanged*
    ELSE:
        DEST.byte[j] := 0
DEST[MAX_VL-1:VL] := 0
```

VPOPCNTW

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.word[j] := POPCNT(SRC.word[j])
    ELSE IF *merging-masking*:
        *DEST.word[j] remains unchanged*
    ELSE:
        DEST.word[j] := 0
DEST[MAX_VL-1:VL] := 0
```

VPOPCNTD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        IF SRC is broadcast memop:
            t := SRC.dword[0]
        ELSE:
            t := SRC.dword[j]
        DEST.dword[j] := POPCNT(t)
    ELSE IF *merging-masking*:
        *DEST..dword[j] remains unchanged*
    ELSE:
        DEST..dword[j] := 0
DEST[MAX_VL-1:VL] := 0
```

VPOPCNTQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        IF SRC is broadcast memop:
            t := SRC.qword[0]
        ELSE:
            t := SRC.qword[j]
        DEST.qword[j] := POPCNT(t)
    ELSE IF *merging-masking*:
        *DEST..qword[j] remains unchanged*
```

```
    ELSE:
        DEST..qword[j] := 0
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPOPCNTW __m128i _mm_popcnt_epi16(__m128i);
VPOPCNTW __m128i _mm_mask_popcnt_epi16(__m128i, __mmask8, __m128i);
VPOPCNTW __m128i _mm_maskz_popcnt_epi16(__mmask8, __m128i);
VPOPCNTW __m256i _mm256_popcnt_epi16(__m256i);
VPOPCNTW __m256i _mm256_mask_popcnt_epi16(__m256i, __mmask16, __m256i);
VPOPCNTW __m256i _mm256_maskz_popcnt_epi16(__mmask16, __m256i);
VPOPCNTW __m512i _mm512_popcnt_epi16(__m512i);
VPOPCNTW __m512i _mm512_mask_popcnt_epi16(__m512i, __mmask32, __m512i);
VPOPCNTW __m512i _mm512_maskz_popcnt_epi16(__mmask32, __m512i);
VPOPCNTQ __m128i _mm_popcnt_epi64(__m128i);
VPOPCNTQ __m128i _mm_mask_popcnt_epi64(__m128i, __mmask8, __m128i);
VPOPCNTQ __m128i _mm_maskz_popcnt_epi64(__mmask8, __m128i);
VPOPCNTQ __m256i _mm256_popcnt_epi64(__m256i);
VPOPCNTQ __m256i _mm256_mask_popcnt_epi64(__m256i, __mmask8, __m256i);
VPOPCNTQ __m256i _mm256_maskz_popcnt_epi64(__mmask8, __m256i);
VPOPCNTQ __m512i _mm512_popcnt_epi64(__m512i);
VPOPCNTQ __m512i _mm512_mask_popcnt_epi64(__m512i, __mmask8, __m512i);
VPOPCNTQ __m512i _mm512_maskz_popcnt_epi64(__mmask8, __m512i);
VPOPCNTD __m128i _mm_popcnt_epi32(__m128i);
VPOPCNTD __m128i _mm_mask_popcnt_epi32(__m128i, __mmask8, __m128i);
VPOPCNTD __m128i _mm_maskz_popcnt_epi32(__mmask8, __m128i);
VPOPCNTD __m256i _mm256_popcnt_epi32(__m256i);
VPOPCNTD __m256i _mm256_mask_popcnt_epi32(__m256i, __mmask8, __m256i);
VPOPCNTD __m256i _mm256_maskz_popcnt_epi32(__mmask8, __m256i);
VPOPCNTD __m512i _mm512_popcnt_epi32(__m512i);
VPOPCNTD __m512i _mm512_mask_popcnt_epi32(__m512i, __mmask16, __m512i);
VPOPCNTD __m512i _mm512_maskz_popcnt_epi32(__mmask16, __m512i);
VPOPCNTB __m128i _mm_popcnt_epi8(__m128i);
VPOPCNTB __m128i _mm_mask_popcnt_epi8(__m128i, __mmask16, __m128i);
VPOPCNTB __m128i _mm_maskz_popcnt_epi8(__mmask16, __m128i);
VPOPCNTB __m256i _mm256_popcnt_epi8(__m256i);
VPOPCNTB __m256i _mm256_mask_popcnt_epi8(__m256i, __mmask32, __m256i);
VPOPCNTB __m256i _mm256_maskz_popcnt_epi8(__mmask32, __m256i);
VPOPCNTB __m512i _mm512_popcnt_epi8(__m512i);
VPOPCNTB __m512i _mm512_mask_popcnt_epi8(__m512i, __mmask64, __m512i);
VPOPCNTB __m512i _mm512_maskz_popcnt_epi8(__mmask64, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”

VPROLD/VPROLVD/VPROLQ/VPROLVQ—Bit Rotate Left

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	Rotate quadwords in zmm2 left by count in the corresponding element of zmm3/m512/m64bcst. Result written to zmm1under writemask k1.
EVEX.512.66.0F.W1 72 /1 ib VPROLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1[1]	Rotate quadwords in zmm2/m512/m64bcst left by imm8. Result written to zmm1 using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	VEX.vvvv (w)	ModRM:r/m (R)	imm8	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

Operation

```
LEFT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 32;
DEST[31:0] := (SRC << COUNT) | (SRC >> (32 - COUNT));

LEFT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 64;
DEST[63:0] := (SRC << COUNT) | (SRC >> (64 - COUNT));
```

VPROLD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPROLVD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[31:0])
                ELSE DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[i+31:i])
            FI;
```

```
        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 Table 2-49, “Type E4 Class Exception Conditions.”

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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	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 OR AVX10.1[1]	Rotate quadwords in zmm2/m512/m64bcst right by imm8, store result using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	VEX.vvvv (w)	ModRM:r/m (R)	imm8	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

Operation

```
RIGHT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 32;
DEST[31:0] := (SRC >> COUNT) | (SRC << (32 - COUNT));

RIGHT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 64;
DEST[63:0] := (SRC >> COUNT) | (SRC << (64 - COUNT));
```

VPRORD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] := RIGHT_ROTATE_DWORDS( SRC1[31:0], imm8)
                ELSE DEST[i+31:i] := RIGHT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPRORVD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := RIGHT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[31:0])
                ELSE DEST[i+31:i] := RIGHT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[i+31:i])
            FI;
        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[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[63:0])
                ELSE DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
```

```
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPRORD __m512i _mm512_ror_epi32(__m512i a, int imm);
VPRORD __m512i _mm512_mask_ror_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPRORD __m512i _mm512_maskz_ror_epi32( __mmask16 k, __m512i a, int imm);
VPRORD __m256i _mm256_ror_epi32(__m256i a, int imm);
VPRORD __m256i _mm256_mask_ror_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPRORD __m256i _mm256_maskz_ror_epi32( __mmask8 k, __m256i a, int imm);
VPRORD __m128i _mm_ror_epi32(__m128i a, int imm);
VPRORD __m128i _mm_mask_ror_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPRORD __m128i _mm_maskz_ror_epi32( __mmask8 k, __m128i a, int imm);
VPRORQ __m512i _mm512_ror_epi64(__m512i a, int imm);
VPRORQ __m512i _mm512_mask_ror_epi64(__m512i a, __mmask8 k, __m512i b, int imm);
VPRORQ __m512i _mm512_maskz_ror_epi64(__mmask8 k, __m512i a, int imm);
VPRORQ __m256i _mm256_ror_epi64(__m256i a, int imm);
VPRORQ __m256i _mm256_mask_ror_epi64(__m256i a, __mmask8 k, __m256i b, int imm);
VPRORQ __m256i _mm256_maskz_ror_epi64( __mmask8 k, __m256i a, int imm);
VPRORQ __m128i _mm_ror_epi64(__m128i a, int imm);
VPRORQ __m128i _mm_mask_ror_epi64(__m128i a, __mmask8 k, __m128i b, int imm);
VPRORQ __m128i _mm_maskz_ror_epi64( __mmask8 k, __m128i a, int imm);
VPRORVD __m512i _mm512_rorv_epi32(__m512i a, __m512i cnt);
VPRORVD __m512i _mm512_mask_rorv_epi32(__m512i a, __mmask16 k, __m512i b, __m512i cnt);
VPRORVD __m512i _mm512_maskz_rorv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPRORVD __m256i _mm256_rorv_epi32(__m256i a, __m256i cnt);
VPRORVD __m256i _mm256_mask_rorv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPRORVD __m256i _mm256_maskz_rorv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPRORVD __m128i _mm_rorv_epi32(__m128i a, __m128i cnt);
VPRORVD __m128i _mm_mask_rorv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORVD __m128i _mm_maskz_rorv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPRORVQ __m512i _mm512_rorv_epi64(__m512i a, __m512i cnt);
VPRORVQ __m512i _mm512_mask_rorv_epi64(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPRORVQ __m512i _mm512_maskz_rorv_epi64( __mmask8 k, __m512i a, __m512i cnt);
VPRORVQ __m256i _mm256_rorv_epi64(__m256i a, __m256i cnt);
VPRORVQ __m256i _mm256_mask_rorv_epi64(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPRORVQ __m256i _mm256_maskz_rorv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPRORVQ __m128i _mm_rorv_epi64(__m128i a, __m128i cnt);
VPRORVQ __m128i _mm_mask_rorv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORVQ __m128i _mm_maskz_rorv_epi64(__mmask8 k, __m128i a, __m128i cnt);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, Signed Qword Indices

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EVEX.128.66.0F38.W0 A0 /vsib VPSCATTERDD vm32x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.256.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32x {k1}, ymm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1[1]	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32y {k1}, zmm1	A	V/V	AVX512F OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Using signed qword indices, scatter qword values to memory using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

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)	N/A	N/A

Description

Stores up to 16 elements (8 elements for qword indices) in doubleword vector or 8 elements in quadword vector to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
- If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination ZMM will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.

The instruction will #UD fault if EVEX.Z = 1.

Operation

```
BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement
```

VPSCATTERDD (EVEX encoded versions)

```
(KL, VL)= (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[i+31:i]) * SCALE + DISP] := SRC[i+31:i]
```

```
                k1[j] := 0

    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VPSCATTERDQ (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[k+31:k]) * SCALE + DISP] := SRC[i+63:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VPSCATTERQD (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] := SRC[i+31:i]
            k1[j] := 0

    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VPSCATTERQQ (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[j+63:j]) * SCALE + DISP] := SRC[i+63:i]
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSCATTERDD void _mm512_i32scatter_epi32(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_i32scatter_epi32(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_i32scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDD void _mm512_mask_i32scatter_epi32(void * base, __mmask16 k, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_mask_i32scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_mask_i32scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void _mm512_i32scatter_epi64(void * base, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm256_i32scatter_epi64(void * base, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void _mm_i32scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void _mm512_mask_i32scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm256_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void _mm_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERQD void _mm512_i64scatter_epi32(void * base, __m512i vdx, __m256i a, int scale);
```

VPSCATTERQD void _mm256_i64scatter_epi32(void * base, __m256i vdx, __m128i a, int scale);
VPSCATTERQD void _mm_i64scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERQD void _mm512_mask_i64scatter_epi32(void * base, __mmask8 k, __m512i vdx, __m256i a, int scale);
VPSCATTERQD void _mm256_mask_i64scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m128i a, int scale);
VPSCATTERQD void _mm_mask_i64scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERQQ void _mm512_i64scatter_epi64(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERQQ void _mm256_i64scatter_epi64(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERQQ void _mm_i64scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERQQ void _mm512_mask_i64scatter_epi64(void * base, __mmask8 k, __m512i vdx, __m512i a, int scale);
VPSCATTERQQ void _mm256_mask_i64scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERQQ void _mm_mask_i64scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-61, “Type E12 Class Exception Conditions.”

VPSHLD—Concatenate and Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 70 /r /ib VPSHLDW xmm1{k1}{z}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W1 70 /r /ib VPSHLDW ymm1{k1}{z}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W1 70 /r /ib VPSHLDW zmm1{k1}{z}, zmm2, zmm3/m512, imm8	A	V/V	AVX512_VBMI2 OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.
EVEX.128.66.0F3A.W0 71 /r /ib VPSHLDD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W0 71 /r /ib VPSHLDD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W0 71 /r /ib VPSHLDD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	B	V/V	AVX512_VBMI2 OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.
EVEX.128.66.0F3A.W1 71 /r /ib VPSHLDQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W1 71 /r /ib VPSHLDQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W1 71 /r /ib VPSHLDQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	B	V/V	AVX512_VBMI2 OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

Concatenate packed data, extract result shifted to the left by constant value.

This instruction supports memory fault suppression.

Operation

VPSHLDW DEST, SRC2, SRC3, imm8

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(SRC2.word[j], SRC3.word[j]) << (imm8 & 15)
        DEST.word[j] := tmp.word[1]
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHLDD DEST, SRC2, SRC3, imm8

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.dword[0]
    ELSE:
        tsrc3 := SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(SRC2.dword[j], tsrc3) << (imm8 & 31)
        DEST.dword[j] := tmp.dword[1]
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHLDQ DEST, SRC2, SRC3, imm8

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.qword[0]
    ELSE:
        tsrc3 := SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(SRC2.qword[j], tsrc3) << (imm8 & 63)
        DEST.qword[j] := tmp.qword[1]
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHLDD __m128i _mm_shldi_epi32(__m128i, __m128i, int);
VPSHLDD __m128i _mm_mask_shldi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDD __m128i _mm_maskz_shldi_epi32(__mmask8, __m128i, __m128i, int);
VPSHLDD __m256i _mm256_shldi_epi32(__m256i, __m256i, int);
VPSHLDD __m256i _mm256_mask_shldi_epi32(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLDD __m256i _mm256_maskz_shldi_epi32(__mmask8, __m256i, __m256i, int);
VPSHLDD __m512i _mm512_shldi_epi32(__m512i, __m512i, int);
VPSHLDD __m512i _mm512_mask_shldi_epi32(__m512i, __mmask16, __m512i, __m512i, int);
VPSHLDD __m512i _mm512_maskz_shldi_epi32(__mmask16, __m512i, __m512i, int);
VPSHLDQ __m128i _mm_shldi_epi64(__m128i, __m128i, int);
VPSHLDQ __m128i _mm_mask_shldi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDQ __m128i _mm_maskz_shldi_epi64(__mmask8, __m128i, __m128i, int);
VPSHLDQ __m256i _mm256_shldi_epi64(__m256i, __m256i, int);
VPSHLDQ __m256i _mm256_mask_shldi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLDQ __m256i _mm256_maskz_shldi_epi64(__mmask8, __m256i, __m256i, int);
VPSHLDQ __m512i _mm512_shldi_epi64(__m512i, __m512i, int);
VPSHLDQ __m512i _mm512_mask_shldi_epi64(__m512i, __mmask8, __m512i, __m512i, int);
VPSHLDQ __m512i _mm512_maskz_shldi_epi64(__mmask8, __m512i, __m512i, int);
VPSHLDW __m128i _mm_shldi_epi16(__m128i, __m128i, int);
VPSHLDW __m128i _mm_mask_shldi_epi16(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDW __m128i _mm_maskz_shldi_epi16(__mmask8, __m128i, __m128i, int);
VPSHLDW __m256i _mm256_shldi_epi16(__m256i, __m256i, int);
VPSHLDW __m256i _mm256_mask_shldi_epi16(__m256i, __mmask16, __m256i, __m256i, int);
VPSHLDW __m256i _mm256_maskz_shldi_epi16(__mmask16, __m256i, __m256i, int);
VPSHLDW __m512i _mm512_shldi_epi16(__m512i, __m512i, int);
VPSHLDW __m512i _mm512_mask_shldi_epi16(__m512i, __mmask32, __m512i, __m512i, int);
VPSHLDW __m512i _mm512_maskz_shldi_epi16(__mmask32, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”

VPSHLDV—Concatenate and Variable Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 70 /r VPSHLDVW xmm1{k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W1 70 /r VPSHLDVW ymm1{k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W1 70 /r VPSHLDVW zmm1{k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI2 OR AVX10.1[1]	Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1.
EVEX.128.66.0F38.W0 71 /r VPSHLDVD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W0 71 /r VPSHLDVD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W0 71 /r VPSHLDVD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VBMI2 OR AVX10.1[1]	Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1.
EVEX.128.66.0F38.W1 71 /r VPSHLDVQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W1 71 /r VPSHLDVQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W1 71 /r VPSHLDVQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512_VBMI2 OR AVX10.1[1]	Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Concatenate packed data, extract result shifted to the left by variable value.

This instruction supports memory fault suppression.

Operation

```
FUNCTION concat(a,b):
    IF words:
        d.word[1] := a
        d.word[0] := b
        return d
    ELSE IF dwords:
        q.dword[1] := a
        q.dword[0] := b
        return q
    ELSE IF qwords:
        o.qword[1] := a
        o.qword[0] := b
        return o
```

VPSHLDVW DEST, SRC2, SRC3

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(DEST.word[j], SRC2.word[j]) << (SRC3.word[j] & 15)
        DEST.word[j] := tmp.word[1]
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHLDVD DEST, SRC2, SRC3

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.dword[0]
    ELSE:
        tsrc3 := SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(DEST.dword[j], SRC2.dword[j]) << (tsrc3 & 31)
        DEST.dword[j] := tmp.dword[1]
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

```
VPSHLDVQ DEST, SRC2, SRC3
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.qword[0]
    ELSE:
        tsrc3 := SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(DEST.qword[j], SRC2.qword[j]) << (tsrc3 & 63)
        DEST.qword[j] := tmp.qword[1]
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSHLDVW __m128i _mm_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVW __m128i _mm_mask_shldv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVW __m128i _mm_maskz_shldv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVW __m256i _mm256_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVW __m256i _mm256_mask_shldv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHLDVW __m256i _mm256_maskz_shldv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHLDVQ __m512i _mm512_shldv_epi64(__m512i, __m512i, __m512i);
VPSHLDVQ __m512i _mm512_mask_shldv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHLDVQ __m512i _mm512_maskz_shldv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHLDVW __m128i _mm_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVW __m128i _mm_mask_shldv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVW __m128i _mm_maskz_shldv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVW __m256i _mm256_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVW __m256i _mm256_mask_shldv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHLDVW __m256i _mm256_maskz_shldv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHLDVW __m512i _mm512_shldv_epi16(__m512i, __m512i, __m512i);
VPSHLDVW __m512i _mm512_mask_shldv_epi16(__m512i, __mmask32, __m512i, __m512i);
VPSHLDVW __m512i _mm512_maskz_shldv_epi16(__mmask32, __m512i, __m512i, __m512i);
VPSHLDVD __m128i _mm_shldv_epi32(__m128i, __m128i, __m128i);
VPSHLDVD __m128i _mm_mask_shldv_epi32(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVD __m128i _mm_maskz_shldv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVD __m256i _mm256_shldv_epi32(__m256i, __m256i, __m256i);
VPSHLDVD __m256i _mm256_mask_shldv_epi32(__m256i, __mmask8, __m256i, __m256i);
VPSHLDVD __m256i _mm256_maskz_shldv_epi32(__mmask8, __m256i, __m256i, __m256i);
VPSHLDVD __m512i _mm512_shldv_epi32(__m512i, __m512i, __m512i);
VPSHLDVD __m512i _mm512_mask_shldv_epi32(__m512i, __mmask16, __m512i, __m512i);
VPSHLDVD __m512i _mm512_maskz_shldv_epi32(__mmask16, __m512i, __m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”

VPSHRD—Concatenate and Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 72 /r /ib VPSHRDW xmm1{k1}{z}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W1 72 /r /ib VPSHRDW ymm1{k1}{z}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W1 72 /r /ib VPSHRDW zmm1{k1}{z}, zmm2, zmm3/m512, imm8	A	V/V	AVX512_VBMI2 OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.
EVEX.128.66.0F3A.W0 73 /r /ib VPSHRDD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W0 73 /r /ib VPSHRDD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W0 73 /r /ib VPSHRDD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	B	V/V	AVX512_VBMI2 OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.
EVEX.128.66.0F3A.W1 73 /r /ib VPSHRDQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W1 73 /r /ib VPSHRDQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W1 73 /r /ib VPSHRDQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	B	V/V	AVX512_VBMI2 OR AVX10.1[1]	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

Concatenate packed data, extract result shifted to the right by constant value.

This instruction supports memory fault suppression.

Operation

VPSHRDW DEST, SRC2, SRC3, imm8

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.word[j] := concat(SRC3.word[j], SRC2.word[j]) >> (imm8 & 15)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHRDD DEST, SRC2, SRC3, imm8

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.dword[0]
    ELSE:
        tsrc3 := SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.dword[j] := concat(tsrc3, SRC2.dword[j]) >> (imm8 & 31)
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHRDQ DEST, SRC2, SRC3, imm8

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.qword[0]
    ELSE:
        tsrc3 := SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.qword[j] := concat(tsrc3, SRC2.qword[j]) >> (imm8 & 63)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHRDQ __m128i _mm_shrdi_epi64(__m128i, __m128i, int);
VPSHRDQ __m128i _mm_mask_shrdi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDQ __m128i _mm_maskz_shrdi_epi64(__mmask8, __m128i, __m128i, int);
VPSHRDQ __m256i _mm256_shrdi_epi64(__m256i, __m256i, int);
VPSHRDQ __m256i _mm256_mask_shrdi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPSHRDQ __m256i _mm256_maskz_shrdi_epi64(__mmask8, __m256i, __m256i, int);
VPSHRDQ __m512i _mm512_shrdi_epi64(__m512i, __m512i, int);
VPSHRDQ __m512i _mm512_mask_shrdi_epi64(__m512i, __mmask8, __m512i, __m512i, int);
VPSHRDQ __m512i _mm512_maskz_shrdi_epi64(__mmask8, __m512i, __m512i, int);
VPSHRDD __m128i _mm_shrdi_epi32(__m128i, __m128i, int);
VPSHRDD __m128i _mm_mask_shrdi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDD __m128i _mm_maskz_shrdi_epi32(__mmask8, __m128i, __m128i, int);
VPSHRDD __m256i _mm256_shrdi_epi32(__m256i, __m256i, int);
VPSHRDD __m256i _mm256_mask_shrdi_epi32(__m256i, __mmask8, __m256i, __m256i, int);
VPSHRDD __m256i _mm256_maskz_shrdi_epi32(__mmask8, __m256i, __m256i, int);
VPSHRDD __m512i _mm512_shrdi_epi32(__m512i, __m512i, int);
VPSHRDD __m512i _mm512_mask_shrdi_epi32(__m512i, __mmask16, __m512i, __m512i, int);
VPSHRDD __m512i _mm512_maskz_shrdi_epi32(__mmask16, __m512i, __m512i, int);
VPSHRDW __m128i _mm_shrdi_epi16(__m128i, __m128i, int);
VPSHRDW __m128i _mm_mask_shrdi_epi16(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDW __m128i _mm_maskz_shrdi_epi16(__mmask8, __m128i, __m128i, int);
VPSHRDW __m256i _mm256_shrdi_epi16(__m256i, __m256i, int);
VPSHRDW __m256i _mm256_mask_shrdi_epi16(__m256i, __mmask16, __m256i, __m256i, int);
VPSHRDW __m256i _mm256_maskz_shrdi_epi16(__mmask16, __m256i, __m256i, int);
VPSHRDW __m512i _mm512_shrdi_epi16(__m512i, __m512i, int);
VPSHRDW __m512i _mm512_mask_shrdi_epi16(__m512i, __mmask32, __m512i, __m512i, int);
VPSHRDW __m512i _mm512_maskz_shrdi_epi16(__mmask32, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”

VPSHRDV—Concatenate and Variable Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 72 /r VPSHRDVW xmm1{k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W1 72 /r VPSHRDVW ymm1{k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W1 72 /r VPSHRDVW zmm1{k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI2 OR AVX10.1[1]	Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1.
EVEX.128.66.0F38.W0 73 /r VPSHRDVD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W0 73 /r VPSHRDVD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W0 73 /r VPSHRDVD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VBMI2 OR AVX10.1[1]	Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1.
EVEX.128.66.0F38.W1 73 /r VPSHRDVQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W1 73 /r VPSHRDVQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1[1]	Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W1 73 /r VPSHRDVQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512_VBMI2 OR AVX10.1[1]	Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Concatenate packed data, extract result shifted to the right by variable value.

This instruction supports memory fault suppression.

Operation

VPSHRDVW DEST, SRC2, SRC3

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.word[j] := concat(SRC2.word[j], DEST.word[j]) >> (SRC3.word[j] & 15)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHRDVD DEST, SRC2, SRC3

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.dword[0]
    ELSE:
        tsrc3 := SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.dword[j] := concat(SRC2.dword[j], DEST.dword[j]) >> (tsrc3 & 31)
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHRDVQ DEST, SRC2, SRC3

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.qword[0]
    ELSE:
        tsrc3 := SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.qword[j] := concat(SRC2.qword[j], DEST.qword[j]) >> (tsrc3 & 63)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHRDVQ __m128i _mm_shrdv_epi64(__m128i, __m128i, __m128i);
VPSHRDVQ __m128i _mm_mask_shrdv_epi64(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVQ __m128i _mm_maskz_shrdv_epi64(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVQ __m256i _mm256_shrdv_epi64(__m256i, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_mask_shrdv_epi64(__m256i, __mmask8, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_maskz_shrdv_epi64(__mmask8, __m256i, __m256i, __m256i);
VPSHRDVQ __m512i _mm512_shrdv_epi64(__m512i, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_mask_shrdv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_maskz_shrdv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHRDVD __m128i _mm_shrdv_epi32(__m128i, __m128i, __m128i);
VPSHRDVD __m128i _mm_mask_shrdv_epi32(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVD __m128i _mm_maskz_shrdv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVD __m256i _mm256_shrdv_epi32(__m256i, __m256i, __m256i);
VPSHRDVD __m256i _mm256_mask_shrdv_epi32(__m256i, __mmask8, __m256i, __m256i);
VPSHRDVD __m256i _mm256_maskz_shrdv_epi32(__mmask8, __m256i, __m256i, __m256i);
VPSHRDVD __m512i _mm512_shrdv_epi32(__m512i, __m512i, __m512i);
VPSHRDVD __m512i _mm512_mask_shrdv_epi32(__m512i, __mmask16, __m512i, __m512i);
VPSHRDVD __m512i _mm512_maskz_shrdv_epi32(__mmask16, __m512i, __m512i, __m512i);
VPSHRDVW __m128i _mm_shrdv_epi16(__m128i, __m128i, __m128i);
VPSHRDVW __m128i _mm_mask_shrdv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVW __m128i _mm_maskz_shrdv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVW __m256i _mm256_shrdv_epi16(__m256i, __m256i, __m256i);
VPSHRDVW __m256i _mm256_mask_shrdv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHRDVW __m256i _mm256_maskz_shrdv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHRDVW __m512i _mm512_shrdv_epi16(__m512i, __m512i, __m512i);
VPSHRDVW __m512i _mm512_mask_shrdv_epi16(__m512i, __mmask32, __m512i, __m512i);
VPSHRDVW __m512i _mm512_maskz_shrdv_epi16(__mmask32, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”

VPSHUFBITQMB—Shuffle Bits From Quadword Elements Using Byte Indexes Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, xmm2, xmm3/m128	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1[1]	Extract values in xmm2 using control bits of xmm3/m128 with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, ymm2, ymm3/m256	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1[1]	Extract values in ymm2 using control bits of ymm3/m256 with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, zmm2, zmm3/m512	A	V/V	AVX512_BITALG OR AVX10.1[1]	Extract values in zmm2 using control bits of zmm3/m512 with writemask k2 and leave the result in mask register k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The VPSHUFBITQMB instruction performs a bit gather select using second source as control and first source as data. Each bit uses 6 control bits (2nd source operand) to select which data bit is going to be gathered (first source operand). A given bit can only access 64 different bits of data (first 64 destination bits can access first 64 data bits, second 64 destination bits can access second 64 data bits, etc.).

Control data for each output bit is stored in 8 bit elements of SRC2, but only the 6 least significant bits of each element are used.

This instruction uses write masking (zeroing only). This instruction supports memory fault suppression.

The first source operand is a ZMM register. The second source operand is a ZMM register or a memory location. The destination operand is a mask register.

Operation

VPSHUFBITQMB DEST, SRC1, SRC2

```
(KL, VL) = (16,128), (32,256), (64, 512)
FOR i := 0 TO KL/8-1:           //Qword
    FOR j := 0 to 7:            // Byte
        IF k2[i*8+j] or *no writemask*:
            m := SRC2.qword[i].byte[j] & 0x3F
            k1[i*8+j] := SRC1.qword[i].bit[m]
        ELSE:
            k1[i*8+j] := 0
k1[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFBITQMB __mmask16 _mm_bitshuffle_epi64_mask(__m128i, __m128i);
VPSHUFBITQMB __mmask16 _mm_mask_bitshuffle_epi64_mask(__mmask16, __m128i, __m128i);
VPSHUFBITQMB __mmask32 _mm256_bitshuffle_epi64_mask(__m256i, __m256i);
VPSHUFBITQMB __mmask32 _mm256_mask_bitshuffle_epi64_mask(__mmask32, __m256i, __m256i);
VPSHUFBITQMB __mmask64 _mm512_bitshuffle_epi64_mask(__m512i, __m512i);
VPSHUFBITQMB __mmask64 _mm512_mask_bitshuffle_epi64_mask(__mmask64, __m512i, __m512i);

VPSLLVW/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical

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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Shift quadwords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the left by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSLLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSLLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation

VPSLLVW (EVEX encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := ZeroExtend(SRC1[i+15:i] << SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSLLVD (VEX.128 version)

```
COUNT_0 := SRC2[31 : 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[127 : 96];
IF COUNT_0 < 32 THEN
DEST[31:0] := ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] := 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
DEST[127:96] := ZeroExtend(SRC1[127:96] << COUNT_3);
ELSE
DEST[127:96] := 0;
DEST[MAXVL-1:128] := 0;
```

VPSLLVD (VEX.256 version)

```
COUNT_0 := SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 := SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] := ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] := 0;
    (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
DEST[255:224] := ZeroExtend(SRC1[255:224] << COUNT_7);
ELSE
DEST[255:224] := 0;
DEST[MAXVL-1:256] := 0;
```

VPSLLVD (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] << SRC2[31:0])
                ELSE DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] << SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSLLVQ (VEX.128 version)

```
COUNT_0 := SRC2[63 : 0];
COUNT_1 := SRC2[127 : 64];
IF COUNT_0 < 64THEN
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 < 64THEN
DEST[63:0] := ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] := 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
DEST[255:192] := ZeroExtend(SRC1[255:192] << COUNT_3);
ELSE
DEST[255:192] := 0;
DEST[MAXVL-1:256] := 0;
```

VPSLLVQ (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] << SRC2[63:0])
                ELSE DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] << SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSLLVW __m512i _mm512_sllv_epi16(__m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_mask_sllv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_maskz_sllv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m256i _mm256_mask_sllv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m256i _mm256_maskz_sllv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m128i _mm_mask_sllv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVW __m128i _mm_maskz_sllv_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m512i _mm512_sllv_epi32(__m512i a, __m512i cnt);
VPSLLVD __m512i _mm512_mask_sllv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSLLVD __m512i _mm512_maskz_sllv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPSLLVD __m256i _mm256_mask_sllv_epi32(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m256i _mm256_maskz_sllv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m128i _mm_mask_sllv_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m128i _mm_maskz_sllv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVQ __m512i _mm512_sllv_epi64(__m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_mask_sllv_epi64(__m512i s, __mmask8 k, __m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_maskz_sllv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPSLLVD __m256i _mm256_mask_sllv_epi64(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m256i _mm256_maskz_sllv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m128i _mm_mask_sllv_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m128i _mm_maskz_sllv_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m256i _mm256_sllv_epi32 (__m256i m, __m256i count)
VPSLLVQ __m256i _mm256_sllv_epi64 (__m256i m, __m256i count)

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded VPSLLVD/VPSLLVQ, see Table 2-49, "Type E4 Class Exception Conditions."
EVEX-encoded VPSLLVW, see Exceptions Type E4.nb in Table 2-49, "Type E4 Class Exception Conditions."

VPSRAVW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic

Opcode/ 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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in sign bits using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (word/doublewords/quadword) in the first source operand (the second operand) to the right by the number of bits specified in the count value of respective data elements in the second source operand (the third operand). As the bits in the data elements are shifted right, the empty high-order bits are set to the MSB (sign extension).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination data element is filled with the corresponding sign bit of the source element.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512/256/128 encoded VPSRAVD/W: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX.512/256/128 encoded VPSRAVQ: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation

VPSRAVW (EVEX encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
   i := j * 16
   IF k1[j] OR *no writemask*
       THEN
           COUNT := SRC2[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 < 64
                        THEN    DEST[i+63:i] := SignExtend(SRC1[i+63:i] >> COUNT)
                        ELSE
                            FOR k := 0 TO 63
                                DEST[i+k] := SRC1[i+63]
                            ENDFOR;
                    FI
                ELSE
                    COUNT := SRC2[i+5:i]
                    IF COUNT < 64
                        THEN    DEST[i+63:i] := SignExtend(SRC1[i+63:i] >> COUNT)
                        ELSE
                            FOR k := 0 TO 63
                                DEST[i+k] := SRC1[i+63]
                            ENDFOR;
                    FI
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSRAVD __m512i _mm512_srav_epi32(__m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_mask_srav_epi32(__m512i s, __mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_maskz_srav_epi32(__mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m256i _mm256_srav_epi32(__m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_mask_srav_epi32(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_maskz_srav_epi32(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m128i _mm_srav_epi32(__m128i a, __m128i cnt);
VPSRAVD __m128i _mm_mask_srav_epi32(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m128i _mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m512i _mm512_srav_epi64(__m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_mask_srav_epi64(__m512i s, __mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_maskz_srav_epi64( __mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m256i _mm256_srav_epi64(__m256i a, __m256i cnt);
VPSRAVQ __m256i _mm256_mask_srav_epi64(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m256i _mm256_maskz_srav_epi64( __mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m128i _mm_srav_epi64(__m128i a, __m128i cnt);
```

```
VPSRAVQ __m128i _mm_mask_srav_epi64(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m128i _mm_maskz_srav_epi64( __mmask8 m, __m128i a, __m128i cnt);
VPSRAVW __m512i _mm512_srav_epi16(__m512i a, __m512i cnt);
VPSRAVW __m512i _mm512_mask_srav_epi16(__m512i s, __mmask32 m, __m512i a, __m512i cnt);
VPSRAVW __m512i _mm512_maskz_srav_epi16(__mmask32 m, __m512i a, __m512i cnt);
VPSRAVW __m256i _mm256_srav_epi16(__m256i a, __m256i cnt);
VPSRAVW __m256i _mm256_mask_srav_epi16(__m256i s, __mmask16 m, __m256i a, __m256i cnt);
VPSRAVW __m256i _mm256_maskz_srav_epi16(__mmask16 m, __m256i a, __m256i cnt);
VPSRAVW __m128i _mm_srav_epi16(__m128i a, __m128i cnt);
VPSRAVW __m128i _mm_mask_srav_epi16(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVW __m128i _mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m256i _mm256_srav_epi32 (__m256i m, __m256i count)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

VPSRLVW/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical

Opcode/ Instruction	Op / En	64/32 bit Mode 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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSRLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSRLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation

VPSRLVW (EVEX encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
         THEN DEST[i+15:i] := ZeroExtend(SRC1[i+15:i] >> SRC2[i+15:i])
         ELSE
              IF *merging-masking*                    ; merging-masking
                   THEN *DEST[i+15:i] remains unchanged*
                   ELSE                               ; zeroing-masking
                        DEST[i+15:i] := 0
              FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSRLVD (VEX.128 version)

```
COUNT_0 := SRC2[31 : 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[127 : 96];
IF COUNT_0 < 32 THEN
    DEST[31:0] := ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
    DEST[31:0] := 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
```

```
IF COUNT_3 < 32 THEN
    DEST[127:96] := ZeroExtend(SRC1[127:96] >> COUNT_3);
ELSE
    DEST[127:96] := 0;
DEST[MAXVL-1:128] := 0;
```

VPSRLVD (VEX.256 version)

```
COUNT_0 := SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 := SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] := ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
DEST[31:0] := 0;
    (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
    DEST[255:224] := ZeroExtend(SRC1[255:224] >> COUNT_7);
ELSE
    DEST[255:224] := 0;
DEST[MAXVL-1:256] := 0;
```

VPSRLVD (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] >> SRC2[31:0])
                ELSE DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] >> SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                        ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                    ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSRLVQ (VEX.128 version)

```
COUNT_0 := SRC2[63 : 0];
COUNT_1 := SRC2[127 : 64];
IF COUNT_0 < 64 THEN
    DEST[63:0] := ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
    DEST[63:0] := 0;
IF COUNT_1 < 64 THEN
    DEST[127:64] := ZeroExtend(SRC1[127:64] >> COUNT_1);
ELSE
    DEST[127:64] := 0;
DEST[MAXVL-1:128] := 0;
```

VPSRLVQ (VEX.256 version)

```
COUNT_0 := SRC2[63 : 0];
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[255 : 192];
IF COUNT_0 < 64 THEN
DEST[63:0] := ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
DEST[63:0] := 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
    DEST[255:192] := ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
    DEST[255:192] := 0;
DEST[MAXVL-1:256] := 0;
```

VPSRLVQ (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] >> SRC2[63:0])
                ELSE DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] >> SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSRLVW __m512i _mm512_srlv_epi16(__m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_mask_srlv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_maskz_srlv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m256i _mm256_mask_srlv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSRLVW __m256i _mm256_maskz_srlv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSRLVW __m128i _mm_mask_srlv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVW __m128i _mm_maskz_srlv_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSRLVW __m256i _mm256_srlv_epi32 (__m256i m, __m256i count)
VPSRLVD __m512i _mm512_srlv_epi32(__m512i a, __m512i cnt);
VPSRLVD __m512i _mm512_mask_srlv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSRLVD __m512i _mm512_maskz_srlv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPSRLVD __m256i _mm256_mask_srlv_epi32(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSRLVD __m256i _mm256_maskz_srlv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPSRLVD __m128i _mm_mask_srlv_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVD __m128i _mm_maskz_srlv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m512i _mm512_srlv_epi64(__m512i a, __m512i cnt);
VPSRLVQ __m512i _mm512_mask_srlv_epi64(__m512i s, __mmask8 k, __m512i a, __m512i cnt);
VPSRLVQ __m512i _mm512_maskz_srlv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPSRLVQ __m256i _mm256_mask_srlv_epi64(__m256i s, __mmask8 k, __m256i a, __m256i cnt);

VPSRLVQ __m256i _mm256_maskz_srlv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPSRLVQ __m128i _mm_mask_srlv_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m128i _mm_maskz_srlv_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m256i _mm256_srlv_epi64 (__m256i m, __m256i count)
VPSRLVD __m128i _mm_srlv_epi32(__m128i a, __m128i cnt);
VPSRLVQ __m128i _mm_srlv_epi64(__m128i a, __m128i cnt);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded VPSRLVD/Q, see Table 2-49, "Type E4 Class Exception Conditions."

EVEX-encoded VPSRLVW, see Exceptions Type E4.nb in Table 2-49, "Type E4 Class Exception Conditions."

VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic

Opcode/ Instruction	Op / En	64/32 bit Mode 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

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 1-19 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 1-19. 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 5-11 and Table 5-12 in the Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2C, 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] << 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[63+i:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[63+i:i] := 0
            FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPTERNLOGD __m512i _mm512_ternarylogic_epi32(__m512i a, __m512i b, int imm);
VPTERNLOGD __m512i _mm512_mask_ternarylogic_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m512i _mm512_maskz_ternarylogic_epi32(__mmask m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m256i _mm256_ternarylogic_epi32(__m256i a, __m256i b, int imm);
VPTERNLOGD __m256i _mm256_mask_ternarylogic_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m256i _mm256_maskz_ternarylogic_epi32( __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m128i _mm_ternarylogic_epi32(__m128i a, __m128i b, int imm);
VPTERNLOGD __m128i _mm_mask_ternarylogic_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGD __m128i _mm_maskz_ternarylogic_epi32( __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGQ __m512i _mm512_ternarylogic_epi64(__m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i _mm512_mask_ternarylogic_epi64(__m512i s, __mmask8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i _mm512_maskz_ternarylogic_epi64( __mmask8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m256i _mm256_ternarylogic_epi64(__m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i _mm256_mask_ternarylogic_epi64(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i _mm256_maskz_ternarylogic_epi64( __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGQ __m128i _mm_ternarylogic_epi64(__m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i _mm_mask_ternarylogic_epi64(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i _mm_maskz_ternarylogic_epi64( __mmask8 m, __m128i a, __m128i b, int imm);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”

VPTESTMB/VPTESTMW/VPTESTMD/VPTESTMQ—Logical AND and Set Mask

Opcode/ 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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical AND operation on the first source operand (the second operand) and second source operand (the third operand) and stores the result in the destination operand (the first operand) under the write-mask. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is non-zero; otherwise it is set to 0.

VPTESTMD/VPTESTMQ: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a mask register updated under the writemask.

VPTESTMB/VPTESTMW: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a mask register updated under the writemask.

Operation

VPTESTMB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN    DEST[j] := (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] != 0)? 1 : 0;
        ELSE    DEST[j] = 0                     ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTMW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN    DEST[j] := (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] != 0)? 1 : 0;
        ELSE    DEST[j] = 0                     ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTMD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
```

```
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN DEST[j] := (SRC1[i+31:i] BITWISE AND SRC2[31:0] != 0)? 1 : 0;
                    ELSE DEST[j] := (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] != 0)? 1 : 0;
                FI;
            ELSE    DEST[j] := 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTMQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;
                ELSE DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;
            FI;
        ELSE    DEST[j] := 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPTESTMB __mmask64 _mm512_test_epi8_mask(__m512i a, __m512i b);
VPTESTMB __mmask64 _mm512_mask_test_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_test_epi16_mask(__m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_mask_test_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_test_epi32_mask(__m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_mask_test_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_test_epi64_mask(__m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_mask_test_epi64_mask(__mmask8, __m512i a, __m512i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VPTESTMD/Q: See Table 2-49, “Type E4 Class Exception Conditions.”

VPTESTMB/W: See Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

VPTESTNMB/W/D/Q—Logical NAND and Set

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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512BW) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical NAND operation on the byte/word/doubleword/quadword element of the first source operand (the second operand) with the corresponding element of the second source operand (the third operand) and stores the logical comparison result into each bit of the destination operand (the first operand) according to the writemask k1. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is zero; otherwise it is set to 0.

EVEX encoded VPTESTNMD/Q: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is updated according to the writemask.

EVEX encoded VPTESTNMB/W: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Operation

VPTESTNMB

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j*8
    IF MaskBit(j) OR *no writemask*
        THEN
            DEST[j] := (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] == 0)? 1 : 0
        ELSE DEST[j] := 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTNMW

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j*16
    IF MaskBit(j) OR *no writemask*
        THEN
            DEST[j] := (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] == 0)? 1 : 0
        ELSE DEST[j] := 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTNMD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j*32
    IF MaskBit(j) OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := (SRC1[i+31:i] BITWISE AND SRC2[31:0] == 0)? 1 : 0
                ELSE DEST[j] := (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] == 0)? 1 : 0
            FI
        ELSE DEST[j] := 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTNMQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j*64
    IF MaskBit(j) OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[63:0] == 0)? 1 : 0;
                ELSE DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] == 0)? 1 : 0;
            FI;
        ELSE DEST[j] := 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPTESTNMB __mmask64 _mm512_testn_epi8_mask( __m512i a, __m512i b);
VPTESTNMB __mmask64 _mm512_mask_testn_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTNMB __mmask32 _mm256_testn_epi8_mask(__m256i a, __m256i b);
VPTESTNMB __mmask32 _mm256_mask_testn_epi8_mask(__mmask32, __m256i a, __m256i b);
VPTESTNMB __mmask16 _mm_testn_epi8_mask(__m128i a, __m128i b);
VPTESTNMB __mmask16 _mm_mask_testn_epi8_mask(__mmask16, __m128i a, __m128i b);
VPTESTNMW __mmask32 _mm512_testn_epi16_mask( __m512i a, __m512i b);
VPTESTNMW __mmask32 _mm512_mask_testn_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTESTNMW __mmask16 _mm256_testn_epi16_mask(__m256i a, __m256i b);
VPTESTNMW __mmask16 _mm256_mask_testn_epi16_mask(__mmask16, __m256i a, __m256i b);
VPTESTNMW __mmask8 _mm_testn_epi16_mask(__m128i a, __m128i b);
VPTESTNMW __mmask8 _mm_mask_testn_epi16_mask(__mmask8, __m128i a, __m128i b);
VPTESTNMD __mmask16 _mm512_testn_epi32_mask( __m512i a, __m512i b);
VPTESTNMD __mmask16 _mm512_mask_testn_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTNMD __mmask8 _mm256_testn_epi32_mask(__m256i a, __m256i b);
VPTESTNMD __mmask8 _mm256_mask_testn_epi32_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMD __mmask8 _mm_testn_epi32_mask(__m128i a, __m128i b);
VPTESTNMD __mmask8 _mm_mask_testn_epi32_mask(__mmask8, __m128i a, __m128i b);
VPTESTNMQ __mmask8 _mm512_testn_epi64_mask(__m512i a, __m512i b);
VPTESTNMQ __mmask8 _mm512_mask_testn_epi64_mask(__mmask8, __m512i a, __m512i b);
VPTESTNMQ __mmask8 _mm256_testn_epi64_mask(__m256i a, __m256i b);
VPTESTNMQ __mmask8 _mm256_mask_testn_epi64_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMQ __mmask8 _mm_testn_epi64_mask(__m128i a, __m128i b);
```

VPTESTNMQ __mmask8 _mm_mask_testn_epi64_mask(__mmask8, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VPTESTNMD/VPTESTNMQ: See Table 2-49, “Type E4 Class Exception Conditions.”

VPTESTNMB/VPTESTNMW: See Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

VRANGEPD—Range Restriction Calculation for Packed Pairs of Float64 Values

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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

This instruction calculates 2/4/8 range operation outputs from two sets of packed input double precision floating-point values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 1-52.

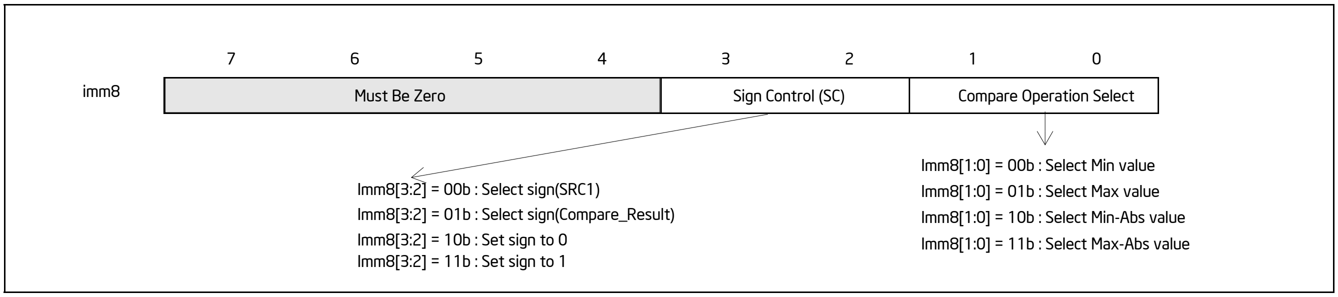


Figure 1-52. Imm8 Controls for VRANGEPD/SD/PS/SS

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 1-20. 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 1-20.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 1-21.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 1-22.

Table 1-20. 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 1-21. 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 1-22. 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 1-20
    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 1-21
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] := from Table 1-22
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[63:0] := (SRC1[63:0] ? SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
        01: TMP[63:0] := (SRC1[63:0] ? SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
        10: TMP[63:0] := (ABS(SRC1[63:0]) ? ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
        11: TMP[63:0] := (ABS(SRC1[63:0]) ? ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
        ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest := (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
    01: dest := TMP[63:0];// Preserve sign of compare result
    10: dest := (0 << 63) OR (TMP[62:0]);// Zero out sign bit
    11: dest := (1 << 63) OR (TMP[62:0]);// Set the sign bit
    ESAC;
    RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGEPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := RangeDP (SRC1[i+63:i], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
                ELSE DEST[i+63:i] := RangeDP (SRC1[i+63:i], SRC2[i+63:i], CmpOpCtl[1:0], SignSelCtl[1:0]);
```

```
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+63:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±1023.

VRANGEPD zmm_dst, zmm_src, zmm_1023, 02h;

Where:

zmm_dst is the destination operand.

zmm_src is the input operand to compare against ±1023 (this is SRC1).

zmm_1023 is the reference operand, contains the value of 1023 (and this is SRC2).

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of SRC1.sign.

In case |zmm_src| < 1023 (i.e., SRC1 is smaller than 1023 in magnitude), then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 1023 (received on zmm_1023, which is SRC2).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of |zmm_src| ? 1023, the selected sign of SRC1 is kept.

Thus, if zmm_src < -1023, the result of VRANGEPD will be the minimal value of -1023 while if zmm_src > +1023, the result of VRANGE will be the maximal value of +1023.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGEPD __m512d _mm512_range_pd (__m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_range_round_pd (__m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d _mm512_mask_range_pd (__m512 ds, __mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_mask_range_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d _mm512_maskz_range_pd (__mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_maskz_range_round_pd (__mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m256d _mm256_range_pd (__m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_mask_range_pd (__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_maskz_range_pd (__mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m128d _mm_range_pd (__m128 a, __m128d b, int imm);
VRANGEPD __m128d _mm_mask_range_pd (__m128 s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGEPD __m128d _mm_maskz_range_pd (__mmask8 k, __m128d a, __m128d b, int imm);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-46, "Type E2 Class Exception Conditions."

VRANGEPS—Range Restriction Calculation for Packed Pairs of Float32 Values

Opcode/ 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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

This instruction calculates 4/8/16 range operation outputs from two sets of packed input single-precision floating-point values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 1-52.

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 1-20. 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 1-20.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 1-21.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 1-22.

Operation

```
RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 1-20
    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 1-21
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] := from Table 1-22
    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 ±150.

VRANGEPS zmm_dst, zmm_src, zmm_150, 02h;

Where:

zmm_dst is the destination operand.

zmm_src is the input operand to compare against ±150.

zmm_150 is the reference operand, contains the value of 150.

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case |zmm_src| < 150, then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 150 (received on zmm_150).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of |zmm_src| ? 150, the selected sign of SRC1 is kept.

Thus, if zmm_src < -150, the result of VRANGEPS will be the minimal value of -150 while if zmm_src > +150, the result of VRANGE will be the maximal value of +150.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGEPS __m512 _mm512_range_ps (__m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_range_round_ps (__m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 _mm512_mask_range_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_mask_range_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 _mm512_maskz_range_ps (__mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_maskz_range_round_ps (__mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m256 _mm256_range_ps (__m256 a, __m256 b, int imm);
VRANGEPS __m256 _mm256_mask_range_ps (__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m256 _mm256_maskz_range_ps (__mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m128 _mm_range_ps (__m128 a, __m128 b, int imm);
VRANGEPS __m128 _mm_mask_range_ps (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGEPS __m128 _mm_maskz_range_ps (__mmask8 k, __m128 a, __m128 b, int imm);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-46, "Type E2 Class Exception Conditions."

VRANGESD—Range Restriction Calculation From a Pair of Scalar Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 51 /r VRANGESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

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 floating-point values in the low qword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low qword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 1-52.

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 1-20. 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 1-20.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 1-21.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 1-22.

Operation

```
RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 1-20
    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 1-21
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] := from Table 1-22
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
        01: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
        10: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
        11: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
        ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest := (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
    01: dest := TMP[63:0];// Preserve sign of compare result
    10: dest := (0 << 63) OR (TMP[62:0]);// Zero out sign bit
    11: dest := (1 << 63) OR (TMP[62:0]);// Set the sign bit
    ESAC;
    RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGESD

```
IF k1[0] OR *no writemask*
        THEN DEST[63:0] := RangeDP (SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] = 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±1023.

VRANGESD xmm_dst, xmm_src, xmm_1023, 02h;

Where:
xmm_dst is the destination operand.
xmm_src is the input operand to compare against ±1023.
xmm_1023 is the reference operand, contains the value of 1023.
IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case |xmm_src| < 1023, then its value will be written into xmm_dst. Otherwise, the value stored in xmm_dst will get the value of 1023 (received on xmm_1023).
However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from xmm_src. So, even in the case of |xmm_src| ≥ 1023, the selected sign of SRC1 is kept.
Thus, if xmm_src < -1023, the result of VRANGEPD will be the minimal value of -1023while if xmm_src > +1023, the result of VRANGE will be the maximal value of +1023.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGESD __m128d _mm_range_sd (__m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_range_round_sd (__m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_mask_range_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_mask_range_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_maskz_range_sd (__mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_maskz_range_round_sd (__mmask8 k, __m128d a, __m128d b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-47, "Type E3 Class Exception Conditions."

VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 51 /r VRANGESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction calculates a range operation output from two input single-precision floating-point values in the low dword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low dword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 1-52.

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 1-20. 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 1-20.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 1-21.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 1-22.

Operation

```
RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 1-20
    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 1-21
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] := from Table 1-22
    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 ±150.

VRANGESS zmm_dst, zmm_src, zmm_150, 02h;

Where:
xmm_dst is the destination operand.
xmm_src is the input operand to compare against ±150.
xmm_150 is the reference operand, contains the value of 150.
IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case |xmm_src| < 150, then its value will be written into zmm_dst. Otherwise, the value stored in xmm_dst will get the value of 150 (received on zmm_150).
However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from xmm_src. So, even in the case of |xmm_src| ≥ 150, the selected sign of SRC1 is kept.
Thus, if xmm_src < -150, the result of VRANGESS will be the minimal value of -150 while if xmm_src > +150, the result of VRANGE will be the maximal value of +150.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGESS __m128 _mm_range_ss (__m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_range_round_ss (__m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_mask_range_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_mask_range_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_maskz_range_ss (__mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_maskz_range_round_ss (__mmask8 k, __m128 a, __m128 b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-47, "Type E3 Class Exception Conditions."

VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values

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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Computes the approximate reciprocals of the packed double precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals of eight/four/two packed double precision floating-point values in the source operand (the second operand) and stores the packed double precision floating-point results in the destination operand. The maximum relative error for this approximation is less than 2^{-14}.

The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ 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 1-23. VRCP14PD/VRCP14SD Special Cases

Input value	Result value	Comments
$0 \le X \le 2^{-1024}$	INF	Very small denormal
$-2^{-1024} \le X \le -0$	-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 Table 2-49, “Type E4 Class Exception Conditions.”

VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W1 4D /r VRCP14SD xmm1 {k1}{z}, xmm2, xmm3/m64	A	V/V	AVX512F OR AVX10.1[1]	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].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocal of the low double precision floating-point value in the second source operand (the third operand) stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register.

The VRCP14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ 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 1-23 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at:

https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP14SD (EVEX version)

```
IF k1[0] OR *no writemask*
        THEN DEST[63:0] := APPROXIMATE(1.0/SRC2[63:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14SD __m128d _mm_rcp14_sd(__m128d a, __m128d b);
VRCP14SD __m128d _mm_mask_rcp14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRCP14SD __m128d _mm_maskz_rcp14_sd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, “Type E5 Class Exception Conditions.”

VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Computes the approximate reciprocals of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals of the packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand). The maximum relative error for this approximation is less than 2^{-14}.

The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ 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 1-24. VRCP14PS/VRCP14SS Special Cases

Input value	Result value	Comments
$0 \le X \le 2^{-128}$	INF	Very small denormal
$-2^{-128} \le X \le -0$	-INF	Very small denormal
$X > 2^{126}$	Underflow	Up to 18 bits of fractions are returned[1]
$X < -2^{126}$	-Underflow	Up to 18 bits of fractions are returned[1]
$X = 2^{-n}$	2^{n}	
$X = -2^{-n}$	-2^{n}	

NOTES:

1. In this case, the mantissa is shifted right by one or two bits.

A numerically exact implementation of VRCP14xx can be found at:

https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP14PS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] := APPROXIMATE(1.0/SRC[31:0]);
                ELSE DEST[i+31:i] := APPROXIMATE(1.0/SRC[i+31:i]);
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14PS __m512 _mm512_rcp14_ps(__m512 a);
VRCP14PS __m512 _mm512_mask_rcp14_ps(__m512 s, __mmask16 k, __m512 a);
VRCP14PS __m512 _mm512_maskz_rcp14_ps(__mmask16 k, __m512 a);
VRCP14PS __m256 _mm256_rcp14_ps(__m256 a);
VRCP14PS __m256 _mm512_mask_rcp14_ps(__m256 s, __mmask8 k, __m256 a);
VRCP14PS __m256 _mm512_maskz_rcp14_ps(__mmask8 k, __m256 a);
VRCP14PS __m128 _mm_rcp14_ps(__m128 a);
VRCP14PS __m128 _mm_mask_rcp14_ps(__m128 s, __mmask8 k, __m128 a);
VRCP14PS __m128 _mm_maskz_rcp14_ps(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”

VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 4D /r VRCP14SS xmm1 {k1}{z}, xmm2, xmm3/m32	A	V/V	AVX512F OR AVX10.1[1]	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].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocal of the low single-precision floating-point value in the second source operand (the third operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

The VRCP14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ 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 1-24 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP14SS (EVEX version)

```
IF k1[0] OR *no writemask*
        THEN DEST[31:0] := APPROXIMATE(1.0/SRC2[31:0]);
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14SS __m128 _mm_rcp14_ss(__m128 a, __m128 b);
VRCP14SS __m128 _mm_mask_rcp14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRCP14SS __m128 _mm_maskz_rcp14_ss(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, “Type E5 Class Exception Conditions.”

VRCPPH—Compute Reciprocals of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 4C /r VRCPPH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Compute the approximate reciprocals of packed FP16 values in xmm2/m128/m16bcst and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 4C /r VRCPPH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Compute the approximate reciprocals of packed FP16 values in ymm2/m256/m16bcst and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 4C /r VRCPPH zmm1{k1}{z}, zmm2/m512/m16bcst	A	V/V	AVX512-FP16 OR AVX10.1[1]	Compute the approximate reciprocals of packed FP16 values in zmm2/m512/m16bcst and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals of 8/16/32 packed FP16 values in the source operand (the second operand) and stores the packed FP16 results in the destination operand. The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$.

For special cases, see Table 1-25.

Table 1-25. VRCPPH/VRCPSH Special Cases

Input Value	Result Value	Comments
$0 \le X \le 2^{-16}$	INF	Very small denormal
$-2^{-16} \le X \le -0$	–INF	Very small denormal
$X > +\infty$	+0	
$X < -\infty$	–0	
$X = 2^{-n}$	2^{n}	
$X = -2^{-n}$	-2^{n}	

Operation

```
VRCPPH dest{k1}, src
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := APPROXIMATE(1.0 / tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRCPPH __m128h _mm_mask_rcp_ph (__m128h src, __mmask8 k, __m128h a);
VRCPPH __m128h _mm_maskz_rcp_ph (__mmask8 k, __m128h a);
VRCPPH __m128h _mm_rcp_ph (__m128h a);
VRCPPH __m256h _mm256_mask_rcp_ph (__m256h src, __mmask16 k, __m256h a);
VRCPPH __m256h _mm256_maskz_rcp_ph (__mmask16 k, __m256h a);
VRCPPH __m256h _mm256_rcp_ph (__m256h a);
VRCPPH __m512h _mm512_mask_rcp_ph (__m512h src, __mmask32 k, __m512h a);
VRCPPH __m512h _mm512_maskz_rcp_ph (__mmask32 k, __m512h a);
VRCPPH __m512h _mm512_rcp_ph (__m512h a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

VRCPSH—Compute Reciprocal of Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 4D /r VRCPSH xmm1{k1}{z}, xmm2, xmm3/m16	A	V/V	AVX512-FP16 OR AVX10.1[1]	Compute the approximate reciprocal of the low FP16 value in xmm3/m16 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocal of the low FP16 value in the second source operand (the third operand) and stores the result in the low word element of the destination operand (the first operand) according to the writemask k1. Bits 127:16 of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

For special cases, see Table 1-25.

Operation

VRCPSH dest{k1}, src1, src2

```
IF k1[0] or *no writemask*:
    DEST.fp16[0] := APPROXIMATE(1.0 / src2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRCPSH __m128h _mm_mask_rcp_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VRCPSH __m128h _mm_maskz_rcp_sh (__mmask8 k, __m128h a, __m128h b);
VRCPSH __m128h _mm_rcp_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-58, “Type E10 Class Exception Conditions.”

VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values

Opcode/ 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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Perform reduction transformation of the packed binary encoded double precision floating-point values in the source operand (the second operand) and store the reduced results in binary floating-point format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 1-53. Specifically, the reduction transformation can be expressed as:

$dest = src – (ROUND(2^{M}*src))*2^{-M};$

where “Round()” treats “src”, “2^{M}”, and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering $src= 2^{p}*man2$,
where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent

Then if RC = RNE: $0<=|Reduced Result|<=2^{p-M-1}$

Then if RC ? RNE: $0<=|Reduced Result|<2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

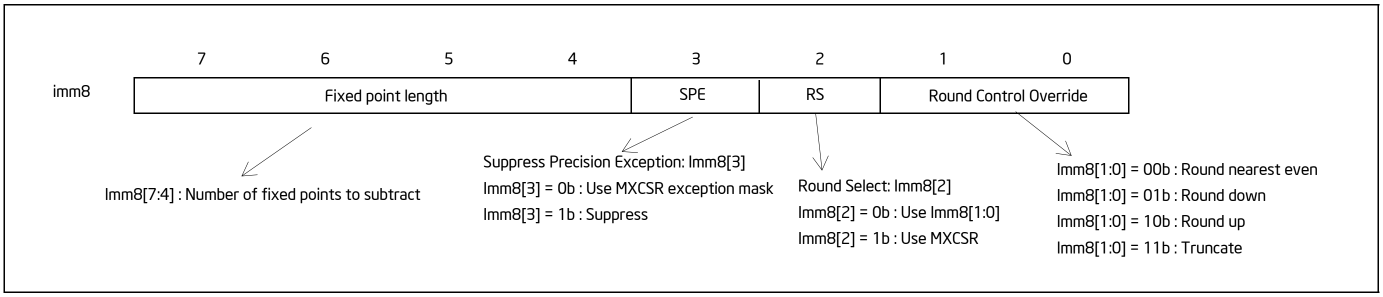


Figure 1-53. Imm8 Controls for VREDUCEPD/SD/PS/SS

Handling of special case of input values are listed in Table 1-26.

Table 1-26. VREDUCEPD/SD/PS/SS Special Cases

	Round Mode	**Returned value**
$\lvert Src1\rvert < 2^{-M-1}$	RNE	Src1
$\lvert Src1\rvert < 2^{-M}$	RPI, Src1 > 0	Round (Src1-2^{-M}) *
	RPI, Src1 ? 0	Src1
	RNI, Src1 ? 0	Src1
	RNI, Src1 < 0	Round (Src1+2^{-M}) *
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 := imm[3];// Suppress Precision Exception
    TMP[63:0] := $2^{-M}$ *{ROUND($2^{M}$*SRC[63:0], SPE, RC_source, RC)}; // ROUND() treats SRC and $2^{M}$ as standard binary FP values
    TMP[63:0] := SRC[63:0] – TMP[63:0]; // subtraction under the same RC,SPE controls
    RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```

```
VREDUCEPD
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] := ReduceArgumentDP(SRC[63:0], imm8[7:0]);
                ELSE DEST[i+63:i] := ReduceArgumentDP(SRC[i+63:i], imm8[7:0]);
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+63:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCEPD __m512d _mm512_mask_reduce_pd(__m512d a, int imm, int sae)
VREDUCEPD __m512d _mm512_mask_reduce_pd(__m512d s, __mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m512d _mm512_maskz_reduce_pd(__mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m256d _mm256_mask_reduce_pd(__m256d a, int imm)
VREDUCEPD __m256d _mm256_mask_reduce_pd(__m256d s, __mmask8 k, __m256d a, int imm)
VREDUCEPD __m256d _mm256_maskz_reduce_pd(__mmask8 k, __m256d a, int imm)
VREDUCEPD __m128d _mm_mask_reduce_pd(__m128d a, int imm)
VREDUCEPD __m128d _mm_mask_reduce_pd(__m128d s, __mmask8 k, __m128d a, int imm)
VREDUCEPD __m128d _mm_maskz_reduce_pd(__mmask8 k, __m128d a, int imm)

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-46, "Type E2 Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B.

VREDUCEPH—Perform Reduction Transformation on Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.0F3A.W0 56 /r /ib VREDUCEPH xmm1{k1}{z}, xmm2/m128/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Perform reduction transformation on packed FP16 values in xmm2/m128/m16bcst by subtracting a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1.
EVEX.256.NP.0F3A.W0 56 /r /ib VREDUCEPH ymm1{k1}{z}, ymm2/m256/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Perform reduction transformation on packed FP16 values in ymm2/m256/m16bcst by subtracting a number of fraction bits specified by the imm8 field. Store the result in ymm1 subject to writemask k1.
EVEX.512.NP.0F3A.W0 56 /r /ib VREDUCEPH zmm1{k1}{z}, zmm2/m512/m16bcst {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1[1]	Perform reduction transformation on packed FP16 values in zmm2/m512/m16bcst by subtracting a number of fraction bits specified by the imm8 field. Store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8 (r)	N/A

Description

This instruction performs a reduction transformation of the packed binary encoded FP16 values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4]. Specifically, the reduction transformation can be expressed as:

$dest = src\ ?\ (ROUND(2^{M} * src)) * 2^{?M}$

where ROUND() treats src, 2^{M}, and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering $src = 2^{p} * man2$, where 'man2' is the normalized significand and 'p' is the unbiased exponent.

Then if RC=RNE: $0\ ?\ |ReducedResult|\ ?\ 2^{?M?1}$.

Then if RC ≠ RNE: $0\ ?\ |ReducedResult| < 2^{?M}$.

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

This instruction may generate tiny non-zero result. If it does so, it does not report underflow exception, even if underflow exceptions are unmasked (UM flag in MXCSR register is 0).

For special cases, see Table 1-27.

Table 1-27. VREDUCEPH/VREDUCESH Special Cases

Input value	Round Mode	Returned Value
$\vert Src1\vert < 2^{?M?1}$	RNE	Src1
$\vert Src1\vert < 2^{?M}$	RU, Src1 > 0	Round(Src1 ? $2^{?M}$)[1]
	RU, Src1 ? 0	Src1
	RD, Src1 ? 0	Src1
	RD, Src1 < 0	Round(Src1 + $2^{?M}$)
Src1 = ±0 or Dest = ±0 (Src1 ≠ ?)	NOT RD	+0.0
	RD	?0.0
Src1 = ±?	Any	+0.0
Src1 = ±NAN	Any	QNaN (Src1)

NOTES:

1. The Round(.) function uses rounding controls specified by (imm8[2]? MXCSR.RC: imm8[1:0]).

Operation

```
def reduce_fp16(src, imm8):
    nan := (src.exp = 0x1F) and (src.fraction != 0)
    if nan:
        return QNAN(src)
    m := imm8[7:4]
    rc := imm8[1:0]
    rc_source := imm8[2]
    spe := imm[3] // suppress precision exception
    tmp := 2^(-m) * ROUND(2^m * src, spe, rc_source, rc)
    tmp := src - tmp // using same RC, SPE controls
    return tmp
```

VREDUCEPH dest{k1}, src, imm8

```
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := reduce_fp16(tsrc, imm8)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCEPH __m128h _mm_mask_reduce_ph (__m128h src, __mmask8 k, __m128h a, int imm8);
VREDUCEPH __m128h _mm_maskz_reduce_ph (__mmask8 k, __m128h a, int imm8);
VREDUCEPH __m128h _mm_reduce_ph (__m128h a, int imm8);
VREDUCEPH __m256h _mm256_mask_reduce_ph (__m256h src, __mmask16 k, __m256h a, int imm8);
VREDUCEPH __m256h _mm256_maskz_reduce_ph (__mmask16 k, __m256h a, int imm8);
VREDUCEPH __m256h _mm256_reduce_ph (__m256h a, int imm8);
VREDUCEPH __m512h _mm512_mask_reduce_ph (__m512h src, __mmask32 k, __m512h a, int imm8);
VREDUCEPH __m512h _mm512_maskz_reduce_ph (__mmask32 k, __m512h a, int imm8);
VREDUCEPH __m512h _mm512_reduce_ph (__m512h a, int imm8);
VREDUCEPH __m512h _mm512_mask_reduce_round_ph (__m512h src, __mmask32 k, __m512h a, int imm8, const int sae);
VREDUCEPH __m512h _mm512_maskz_reduce_round_ph (__mmask32 k, __m512h a, int imm8, const int sae);
VREDUCEPH __m512h _mm512_reduce_round_ph (__m512h a, int imm8, const int sae);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”

VREDUCEPS—Perform Reduction Transformation on Packed Float32 Values

Opcode/ 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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Perform reduction transformation of the packed binary encoded single-precision floating-point values in the source operand (the second operand) and store the reduced results in binary floating-point format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 1-53. Specifically, the reduction transformation can be expressed as:

dest = src – (ROUND(2^M*src))*2^{-M};

where “Round()” treats “src”, “2^M”, and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2,
where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent

Then if RC = RNE: 0<=|Reduced Result|<=2^{p-M-1}

Then if RC ? RNE: 0<=|Reduced Result|<2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Handling of special case of input values are listed in Table 1-26.

Operation

```
ReduceArgumentSP(SRC[31:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [31:0] = NAN) THEN
        RETURN (Convert SRC[31:0] to QNaN); FI
    M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC := imm8[1:0];// Round Control for ROUND() operation
    RC source := imm[2];
    SPE := imm[3];// Suppress Precision Exception
    TMP[31:0] := $2^{-M}$ *{ROUND($2^{M}$*SRC[31:0], SPE, RC_source, RC)}; // ROUND() treats SRC and $2^{M}$ as standard binary FP values
    TMP[31:0] := SRC[31:0] – TMP[31:0]; // subtraction under the same RC,SPE controls
RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}
```

VREDUCEPS

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b == 1) AND (SRC *is memory*)
                    THEN DEST[i+31:i] := ReduceArgumentSP(SRC[31:0], imm8[7:0]);
                    ELSE DEST[i+31:i] := ReduceArgumentSP(SRC[i+31:i], imm8[7:0]);
                FI;
    ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] = 0
            FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCEPS __m512 _mm512_mask_reduce_ps(__m512 a, int imm, int sae)
VREDUCEPS __m512 _mm512_mask_reduce_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m512 _mm512_maskz_reduce_ps(__mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m256 _mm256_mask_reduce_ps(__m256 a, int imm)
VREDUCEPS __m256 _mm256_mask_reduce_ps(__m256 s, __mmask8 k, __m256 a, int imm)
VREDUCEPS __m256 _mm256_maskz_reduce_ps(__mmask8 k, __m256 a, int imm)
VREDUCEPS __m128 _mm_mask_reduce_ps(__m128 a, int imm)
VREDUCEPS __m128 _mm_mask_reduce_ps(__m128 s, __mmask8 k, __m128 a, int imm)
VREDUCEPS __m128 _mm_maskz_reduce_ps(__mmask8 k, __m128 a, int imm)

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions”; additionally:

#UD	If EVEX.vvvv != 1111B.

VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 57 VREDUCESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8/r	A	V/V	AVX512DQ OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Perform a reduction transformation of the binary encoded double precision floating-point value in the low qword element of the second source operand (the third operand) and store the reduced result in binary floating-point format to the low qword element of the destination operand (the first operand) under the writemask k1. Bits 127:64 of the destination operand are copied from respective qword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 1-53. Specifically, the reduction transformation can be expressed as:

$dest = src – (ROUND(2^{M}*src))*2^{-M}$;

where “Round()” treats “src”, “2^{M}”, and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= $2^{p}*man2$,
where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent

Then if RC = RNE: $0<=|Reduced Result|<=2^{p-M-1}$

Then if RC ≠ RNE: $0<=|Reduced Result|<2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

The operation is write masked.

Handling of special case of input values are listed in Table 1-26.

Operation

```
ReduceArgumentDP(SRC[63:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [63:0] = NAN) THEN
        RETURN (Convert SRC[63:0] to QNaN); FI;
    M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC := imm8[1:0];// Round Control for ROUND() operation
    RC source := imm[2];
    SPE := imm[3];// Suppress Precision Exception
```

```
    TMP[63:0] := $2^{-M}$ *{ROUND($2^{M}$*SRC[63:0], SPE, RC_source, RC)}; // ROUND() treats SRC and $2^{M}$ as standard binary FP values
    TMP[63:0] := SRC[63:0] – TMP[63:0]; // subtraction under the same RC,SPE controls
    RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```

VREDUCESD

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := ReduceArgumentDP(SRC2[63:0], imm8[7:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] = 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCESD __m128d _mm_mask_reduce_sd(__m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_mask_reduce_sd(__m128d s, __mmask16 k, __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_maskz_reduce_sd(__mmask16 k, __m128d a, __m128d b, int imm, int sae)

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions.”

VREDUCESH—Perform Reduction Transformation on Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.0F3A.W0 57 /r /ib VREDUCESH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1[1]	Perform a reduction transformation on the low binary encoded FP16 value in xmm3/m16 by subtracting a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

This instruction performs a reduction transformation of the low binary encoded FP16 value in the source operand (the second operand) and store the reduced result in binary FP format to the low element of the destination operand (the first operand) under the writemask k1. For further details see the description of VREDUCEPH.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

This instruction may generate tiny non-zero result. If it does so, it does not report underflow exception, even if underflow exceptions are unmasked (UM flag in MXCSR register is 0).

For special cases, see Table 1-27.

Operation

VREDUCESH dest{k1}, src, imm8

```
IF k1[0] or *no writemask*:
    dest.fp16[0] := reduce_fp16(src2.fp16[0], imm8)     // see VREDUCEPH
ELSE IF *zeroing*:
    dest.fp16[0] := 0
//else dest.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VREDUCESH __m128h _mm_mask_reduce_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8, const int sae);
VREDUCESH __m128h _mm_maskz_reduce_round_sh (__mmask8 k, __m128h a, __m128h b, int imm8, const int sae);
VREDUCESH __m128h _mm_reduce_round_sh (__m128h a, __m128h b, int imm8, const int sae);
VREDUCESH __m128h _mm_mask_reduce_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8);
VREDUCESH __m128h _mm_maskz_reduce_sh (__mmask8 k, __m128h a, __m128h b, int imm8);
VREDUCESH __m128h _mm_reduce_sh (__m128h a, __m128h b, int imm8);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VREDUCESS—Perform a Reduction Transformation on a Scalar Float32 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 57 /r /ib VREDUCESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Perform a reduction transformation of the binary encoded single-precision floating-point value in the low dword element of the second source operand (the third operand) and store the reduced result in binary floating-point format to the low dword element of the destination operand (the first operand) under the writemask k1. Bits 127:32 of the destination operand are copied from respective dword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 1-53. Specifically, the reduction transformation can be expressed as:

$dest = src – (ROUND(2^{M}*src))*2^{-M};$

where "Round()" treats "src", "2^M", and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^{p}*man2,
where 'man2' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE: $0<=|Reduced Result|<=2^{p-M-1}$

Then if RC ≠ RNE: $0<=|Reduced Result|<2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

Handling of special case of input values are listed in Table 1-26.

Operation

```
ReduceArgumentSP(SRC[31:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [31:0] = NAN) THEN
        RETURN (Convert SRC[31:0] to QNaN); FI
    M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC := imm8[1:0];// Round Control for ROUND() operation
    RC source := imm[2];
    SPE := imm[3];// Suppress Precision Exception
```

```
    TMP[31:0] := $2^{-M}$ *{ROUND($2^M$*SRC[31:0], SPE, RC_source, RC)}; // ROUND() treats SRC and $2^M$ as standard binary FP values
    TMP[31:0] := SRC[31:0] – TMP[31:0]; // subtraction under the same RC,SPE controls
RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}
```

VREDUCESS

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := ReduceArgumentSP(SRC2[31:0], imm8[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] = 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCESS __m128 _mm_mask_reduce_ss(__m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 _mm_mask_reduce_ss(__m128 s, __mmask16 k, __m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 _mm_maskz_reduce_ss(__mmask16 k, __m128 a, __m128 b, int imm, int sae)

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions.”

VRNDSCALEPD—Round Packed Float64 Values to Include a Given Number of Fraction Bits

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Round the double precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 1-54) 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

$\mathrm{ROUND}(x) = 2^{-M}*\mathrm{Round\_to\_INT}(x*2^M, \mathrm{round\_ctrl}),$

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of $x*2^M$ is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALEPD is a more general form of the VEX-encoded VROUNDPD instruction. In VROUNDPD, the formula of the operation on each element is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = imm[3:0];

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

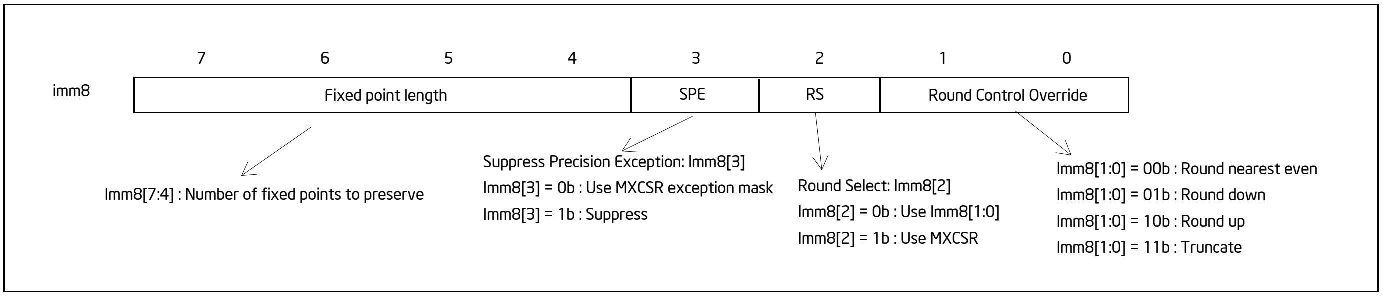


Figure 1-54. Imm8 Controls for VRNDSCALEPD/SD/PS/SS

Handling of special case of input values are listed in Table 1-28.

Table 1-28. VRNDSCALEPD/SD/PS/SS Special Cases

	Returned value
Src1=±inf	Src1
Src1=±NAN	Src1 converted to QNAN
Src1=±0	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($2^M$*SRC[63:0])
    01: TMP[63:0] := round_to_equal_or_smaller_integer($2^M$*SRC[63:0])
    10: TMP[63:0] := round_to_equal_or_larger_integer($2^M$*SRC[63:0])
    11: TMP[63:0] := round_to_nearest_smallest_magnitude_integer($2^M$*SRC[63:0])
    ESAC

    Dest[63:0] := $2^{-M}$* TMP[63:0]           ; scale down back to $2^{-M}$

    if (imm8[3] = 0) Then    ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then       ; check precision lost
            set_precision()                     ; set #PE
        FI;
    FI;
```

```
    return(Dest[63:0])
}
```

VRNDSCALEPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF *src is a memory operand*
    THEN TMP_SRC := BROADCAST64(SRC, VL, k1)
    ELSE TMP_SRC := SRC
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := RoundToIntegerDP((TMP_SRC[i+63:i], imm8[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+63:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRNDSCALEPD __m512d _mm512_roundscale_pd( __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_roundscale_round_pd( __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_mask_roundscale_pd(__m512d s, __mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_mask_roundscale_round_pd(__m512d s, __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_pd( __mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_round_pd( __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m256d _mm256_roundscale_pd( __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_mask_roundscale_pd(__m256d s, __mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_maskz_roundscale_pd( __mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m128d _mm_roundscale_pd( __m128d a, int imm);
VRNDSCALEPD __m128d _mm_mask_roundscale_pd(__m128d s, __mmask8 k, __m128d a, int imm);
VRNDSCALEPD __m128d _mm_maskz_roundscale_pd( __mmask8 k, __m128d a, int imm);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”

VRNDSCALEPH—Round Packed FP16 Values to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.0F3A.W0 08 /r /ib VRNDSCALEPH xmm1{k1}{z}, xmm2/m128/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Round packed FP16 values in xmm2/m128/m16bcst to a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1.
EVEX.256.NP.0F3A.W0 08 /r /ib VRNDSCALEPH ymm1{k1}{z}, ymm2/m256/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Round packed FP16 values in ymm2/m256/m16bcst to a number of fraction bits specified by the imm8 field. Store the result in ymm1 subject to writemask k1.
EVEX.512.NP.0F3A.W0 08 /r /ib VRNDSCALEPH zmm1{k1}{z}, zmm2/m512/m16bcst {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1[1]	Round packed FP16 values in zmm2/m512/m16bcst to a number of fraction bits specified by the imm8 field. Store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8 (r)	N/A

Description

This instruction rounds the FP16 values in the source operand by the rounding mode specified in the immediate operand (see Table 1-29) and places the result in the destination operand. The destination operand is conditionally updated according to the writemask.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result), and returns the result as an FP16 value.

Note that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation. Three bit fields are defined and shown in Table 1-29, "Imm8 Controls for VRNDSCALEPH/VRNDSCALESH." Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control, and bits 1:0 specify a non-sticky rounding-mode value.

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN.

The sign of the result of this instruction is preserved, including the sign of zero. Special cases are described in Table 1-30.

The formula of the operation on each data element for VRNDSCALEPH is

ROUND(x) = $2^{?M}$ *Round_to_INT(x * 2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x * 2^{M} is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

If this instruction encoding's SPE bit (bit 3) in the immediate operand is 1, VRNDSCALEPH can set MXCSR.UE without MXCSR.PE.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Table 1-29. Imm8 Controls for VRNDSCALEPH/VRNDSCALESH

Imm8 Bits	Description
imm8[7:4]	Number of fixed points to preserve.
imm8[3]	Suppress Precision Exception (SPE) 0b00: Implies use of MXCSR exception mask. 0b01: Implies suppress.
imm8[2]	Round Select (RS) 0b00: Implies use of imm8[1:0]. 0b01: Implies use of MXCSR.
imm8[1:0]	Round Control Override: 0b00: Round nearest even. 0b01: Round down. 0b10: Round up. 0b11: Truncate.

Table 1-30. VRNDSCALEPH/VRNDSCALESH Special Cases

Input Value	Returned Value
Src1 = ±?	Src1
Src1 = ±NaN	Src1 converted to QNaN
Src1 = ±0	Src1

Operation

```
def round_fp16_to_integer(src, imm8):
    if imm8[2] = 1:
        rounding_direction := MXCSR.RC
    else:
        rounding_direction := imm8[1:0]
    m := imm8[7:4] // scaling factor

    tsrc1 := 2^m * src

    if rounding_direction = 0b00:
        tmp := round_to_nearest_even_integer(trc1)
    else if rounding_direction = 0b01:
        tmp := round_to_equal_or_smaller_integer(trc1)
    else if rounding_direction = 0b10:
        tmp := round_to_equal_or_larger_integer(trc1)
    else if rounding_direction = 0b11:
        tmp := round_to_smallest_magnitude_integer(trc1)

    dst := 2^(-m) * tmp

    if imm8[3]==0: // check SPE
        if src != dst:
            MXCSR.PE := 1
    return dst
```

```
VRNDSCALEPH dest{k1}, src, imm8
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := round_fp16_to_integer(tsrc, imm8)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALEPH __m128h _mm_mask_roundscale_ph (__m128h src, __mmask8 k, __m128h a, int imm8);
VRNDSCALEPH __m128h _mm_maskz_roundscale_ph (__mmask8 k, __m128h a, int imm8);
VRNDSCALEPH __m128h _mm_roundscale_ph (__m128h a, int imm8);
VRNDSCALEPH __m256h _mm256_mask_roundscale_ph (__m256h src, __mmask16 k, __m256h a, int imm8);
VRNDSCALEPH __m256h _mm256_maskz_roundscale_ph (__mmask16 k, __m256h a, int imm8);
VRNDSCALEPH __m256h _mm256_roundscale_ph (__m256h a, int imm8);
VRNDSCALEPH __m512h _mm512_mask_roundscale_ph (__m512h src, __mmask32 k, __m512h a, int imm8);
VRNDSCALEPH __m512h _mm512_maskz_roundscale_ph (__mmask32 k, __m512h a, int imm8);
VRNDSCALEPH __m512h _mm512_roundscale_ph (__m512h a, int imm8);
VRNDSCALEPH __m512h _mm512_mask_roundscale_round_ph (__m512h src, __mmask32 k, __m512h a, int imm8, const int sae);
VRNDSCALEPH __m512h _mm512_maskz_roundscale_round_ph (__mmask32 k, __m512h a, int imm8, const int sae);
VRNDSCALEPH __m512h _mm512_roundscale_round_ph (__m512h a, int imm8, const int sae);

SIMD Floating-Point Exceptions

Invalid, Underflow, Precision.

Other Exceptions

EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”

VRNDSCALEPS—Round Packed Float32 Values to Include a Given Number of Fraction Bits

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Round the single-precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 1-54) 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

$\mathrm{ROUND}(x) = 2^{-M}*\mathrm{Round\_to\_INT}(x*2^{M}, \mathrm{round\_ctrl}),$

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of $x*2^{M}$ is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALEPS is a more general form of the VEX-encoded VROUNDPS instruction. In VROUNDPS, the formula of the operation on each element is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = imm[3:0];

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Handling of special case of input values are listed in Table 1-28.

Operation

```
RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction := MXCSR:RC          ; get round control from MXCSR
    else
        rounding_direction := imm8[1:0]         ; get round control from imm8[1:0]
    FI
    M := imm8[7:4]              ; get the scaling factor

    case (rounding_direction)
    00: TMP[31:0] := round_to_nearest_even_integer($2^{M}$*SRC[31:0])
    01: TMP[31:0] := round_to_equal_or_smaller_integer($2^{M}$*SRC[31:0])
    10: TMP[31:0] := round_to_equal_or_larger_integer($2^{M}$*SRC[31:0])
    11: TMP[31:0] := round_to_nearest_smallest_magnitude_integer($2^{M}$*SRC[31:0])
    ESAC;

    Dest[31:0] := $2^{-M}$* TMP[31:0]           ; scale down back to $2^{-M}$
    if (imm8[3] = 0) Then           ; check SPE
        if (SRC[31:0] != Dest[31:0]) Then       ; check precision lost
            set_precision()             ; set #PE
        FI;
    FI;
    return(Dest[31:0])
}

VRNDSCALEPS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF *src is a memory operand*
    THEN TMP_SRC := BROADCAST32(SRC, VL, k1)
    ELSE TMP_SRC := SRC
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := RoundToIntegerSP(TMP_SRC[i+31:i]), imm8[7:0])
    ELSE
        IF *merging-masking*                        ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                    ; zeroing-masking
                DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALEPS __m512 _mm512_roundscale_ps(__m512 a, int imm);
VRNDSCALEPS __m512 _mm512_roundscale_round_ps(__m512 a, int imm, int sae);
VRNDSCALEPS __m512 _mm512_mask_roundscale_ps(__m512 s, __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 _mm512_mask_roundscale_round_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m512 _mm512_maskz_roundscale_ps(__mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 _mm512_maskz_roundscale_round_ps(__mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m256 _mm256_roundscale_ps(__m256 a, int imm);
VRNDSCALEPS __m256 _mm256_mask_roundscale_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m256 _mm256_maskz_roundscale_ps(__mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m128 _mm_roundscale_ps(__m256 a, int imm);
VRNDSCALEPS __m128 _mm_mask_roundscale_ps(__m128 s, __mmask8 k, __m128 a, int imm);
VRNDSCALEPS __m128 _mm_maskz_roundscale_ps(__mmask8 k, __m128 a, int imm);

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”

VRNDSCALESD—Round Scalar Float64 Value to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 0B /r ib VRNDSCALESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

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 1-54) element of the second source operand (the third operand) by the rounding mode specified in the immediate operand and places the result in the corresponding element of the destination operand (the first operand) according to the writemask. The quadword element at bits 127:64 of the destination is copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the “Immediate Control Description” figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESD is

$\text{ROUND}(x) = 2^{-M}*\text{Round\_to\_INT}(x*2^{M}, \text{round\_ctrl}),$

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of $x*2^{M}$ is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALESD is a more general form of the VEX-encoded VROUNDSD instruction. In VROUNDSD, the formula of the operation is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = imm[3:0];

EVEX encoded version: The source operand is a XMM register or a 64-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 1-28.

Operation

```
RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction := MXCSR:RC          ; get round control from MXCSR
    else
        rounding_direction := imm8[1:0]         ; get round control from imm8[1:0]
    FI
    M := imm8[7:4]          ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0] := round_to_nearest_even_integer($2^M$*SRC[63:0])
    01: TMP[63:0] := round_to_equal_or_smaller_integer($2^M$*SRC[63:0])
    10: TMP[63:0] := round_to_equal_or_larger_integer($2^M$*SRC[63:0])
    11: TMP[63:0] := round_to_nearest_smallest_magnitude_integer($2^M$*SRC[63:0])
    ESAC

    Dest[63:0] := $2^{-M}$* TMP[63:0]          ; scale down back to $2^{-M}$

    if (imm8[3] = 0) Then   ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then       ; check precision lost
            set_precision()                     ; set #PE
        FI;
    FI;
    return(Dest[63:0])
}
```

```
VRNDSCALESD (EVEX encoded version)
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundToIntegerDP(SRC2[63:0], Zero_upper_imm[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRNDSCALESD __m128d _mm_roundscale_sd ( __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_roundscale_round_sd ( __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d _mm_mask_roundscale_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_mask_roundscale_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d _mm_maskz_roundscale_sd ( __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_maskz_roundscale_round_sd ( __mmask8 k, __m128d a, __m128d b, int imm, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions.”

VRNDSCALESH—Round Scalar FP16 Value to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.0F3A.W0 0A /r /ib VRNDSCALESH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1[1]	Round the low FP16 value in xmm3/m16 to a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

This instruction rounds the low FP16 value in the second source operand by the rounding mode specified in the immediate operand (see Table 1-29) and places the result in the destination operand.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result), and returns the result as a FP16 value.

Note that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation. Three bit fields are defined and shown in Table 1-29, “Imm8 Controls for VRNDSCALEPH/VRNDSCALESH.” Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control, and bits 1:0 specify a non-sticky rounding-mode value.

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN.

The sign of the result of this instruction is preserved, including the sign of zero. Special cases are described in Table 1-30.

If this instruction encoding’s SPE bit (bit 3) in the immediate operand is 1, VRNDSCALESH can set MXCSR.UE without MXCSR.PE.

The formula of the operation on each data element for VRNDSCALESH is:

$\text{ROUND}(x) = 2^{-M} * \text{Round\_to\_INT}(x * 2^{M}, \text{round\_ctrl})$,

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of $x * 2^{M}$ is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

Operation

VRNDSCALESH dest{k1}, src1, src2, imm8

```
IF k1[0] or *no writemask*:
    DEST.fp16[0] := round_fp16_to_integer(src2.fp16[0], imm8) // see VRNDSCALEPH
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[127:16] = src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALESH __m128h _mm_mask_roundscale_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8, const int sae);

VRNDSCALESH __m128h _mm_maskz_roundscale_round_sh (__mmask8 k, __m128h a, __m128h b, int imm8, const int sae);

VRNDSCALESH __m128h _mm_roundscale_round_sh (__m128h a, __m128h b, int imm8, const int sae);

VRNDSCALESH __m128h _mm_mask_roundscale_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8);

VRNDSCALESH __m128h _mm_maskz_roundscale_sh (__mmask8 k, __m128h a, __m128h b, int imm8);

VRNDSCALESH __m128h _mm_roundscale_sh (__m128h a, __m128h b, int imm8);

SIMD Floating-Point Exceptions

Invalid, Underflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VRNDSCALESS—Round Scalar Float32 Value to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 0A /r ib VRNDSCALESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Rounds the single-precision floating-point value in the low doubleword element of the second source operand (the third operand) by the rounding mode specified in the immediate operand (see Figure 1-54) and places the result in the corresponding element of the destination operand (the first operand) according to the writemask. The doubleword elements at bits 127:32 of the destination are copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the “Immediate Control Description” figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (immediate control tables below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESS is

ROUND(x) = 2^{-M}*Round_to_INT(x*2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x*2^{M} is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALESS is a more general form of the VEX-encoded VROUNDSS instruction. In VROUNDSS, the formula of the operation on each element is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = 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 1-28.

Operation

```
RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction := MXCSR:RC          ; get round control from MXCSR
    else
        rounding_direction := imm8[1:0]         ; get round control from imm8[1:0]
    FI
    M := imm8[7:4]          ; get the scaling factor

    case (rounding_direction)
    00: TMP[31:0] := round_to_nearest_even_integer($2^M$*SRC[31:0])
    01: TMP[31:0] := round_to_equal_or_smaller_integer($2^M$*SRC[31:0])
    10: TMP[31:0] := round_to_equal_or_larger_integer($2^M$*SRC[31:0])
    11: TMP[31:0] := round_to_nearest_smallest_magnitude_integer($2^M$*SRC[31:0])
    ESAC;

    Dest[31:0] := $2^{-M}$* TMP[31:0]          ; scale down back to $2^{-M}$
    if (imm8[3] = 0) Then          ; check SPE
        if (SRC[31:0] != Dest[31:0]) Then          ; check precision lost
            set_precision()          ; set #PE
        FI;
    FI;
    return(Dest[31:0])
}
```

VRNDSCALESS (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundToIntegerSP(SRC2[31:0], Zero_upper_imm[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALESS __m128 _mm_roundscale_ss (__m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_roundscale_round_ss (__m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_mask_roundscale_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_mask_roundscale_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_maskz_roundscale_ss (__mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_maskz_roundscale_round_ss (__mmask8 k, __m128 a, __m128 b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions.”

VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of the eight packed double precision floating-point values in the source operand (the second operand) and stores the packed double precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}.

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

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 1-31. 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 _mm512_mask_rsqrt14_pd(__m256d s, __mmask8 k, __m256d a);
VRSQRT14PD __m256d _mm512_maskz_rsqrt14_pd( __mmask8 k, __m256d a);
VRSQRT14PD __m128d _mm_rsqrt14_pd( __m128d a);
VRSQRT14PD __m128d _mm_mask_rsqrt14_pd(__m128d s, __mmask8 k, __m128d a);
VRSQRT14PD __m128d _mm_maskz_rsqrt14_pd( __mmask8 k, __m128d a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”

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.LLIG.66.0F38.W1 4F /r VRSQRT14SD xmm1 {k1}{z}, xmm2, xmm3/m64	A	V/V	AVX512F OR AVX10.1[1]	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].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes the approximate reciprocal of the square roots of the scalar double precision floating-point value in the low quadword element of the source operand (the second operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

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 1-32. VRSQRT14SD Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14SD __m128d _mm_rsqrt14_sd(__m128d a, __m128d b);
VRSQRT14SD __m128d _mm_mask_rsqrt14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRSQRT14SD __m128d _mm_maskz_rsqrt14_sd(__mmask8d m, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, “Type E5 Class Exception Conditions.”

VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of 16 packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}.

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

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 1-33. VRSQRT14PS Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14PS __m512 _mm512_rsqrt14_ps(__m512 a);
VRSQRT14PS __m512 _mm512_mask_rsqrt14_ps(__m512 s, __mmask16 k, __m512 a);
VRSQRT14PS __m512 _mm512_maskz_rsqrt14_ps(__mmask16 k, __m512 a);
VRSQRT14PS __m256 _mm256_rsqrt14_ps(__m256 a);
VRSQRT14PS __m256 _mm256_mask_rsqrt14_ps(__m256 s, __mmask8 k, __m256 a);
VRSQRT14PS __m256 _mm256_maskz_rsqrt14_ps(__mmask8 k, __m256 a);
VRSQRT14PS __m128 _mm_rsqrt14_ps(__m128 a);
VRSQRT14PS __m128 _mm_mask_rsqrt14_ps(__m128 s, __mmask8 k, __m128 a);
VRSQRT14PS __m128 _mm_maskz_rsqrt14_ps(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 4F /r VRSQRT14SS xmm1 {k1}{z}, xmm2, xmm3/m32	A	V/V	AVX512F OR AVX10.1[1]	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].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes 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 ∞ with the sign of the source value is returned. When the source operand is an ∞, zero with the sign of the source value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRSQRT14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT14SS (EVEX version)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := APPROXIMATE(1.0/ SQRT(SRC2[31:0]))
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Table 1-34. VRSQRT14SS Special Cases

Input value	**Result value**	**Comments**
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14SS __m128 _mm_rsqrt14_ss(__m128 a, __m128 b);
VRSQRT14SS __m128 _mm_mask_rsqrt14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRSQRT14SS __m128 _mm_maskz_rsqrt14_ss(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, “Type E5 Class Exception Conditions.”

VRSQRTPH—Compute Reciprocals of Square Roots of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 4E /r VRSQRTPH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Compute the approximate reciprocals of the square roots of packed FP16 values in xmm2/m128/m16bcst and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 4E /r VRSQRTPH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Compute the approximate reciprocals of the square roots of packed FP16 values in ymm2/m256/m16bcst and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 4E /r VRSQRTPH zmm1{k1}{z}, zmm2/m512/m16bcst	A	V/V	AVX512-FP16 OR AVX10.1[1]	Compute the approximate reciprocals of the square roots of packed FP16 values in zmm2/m512/m16bcst and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals square-root of 8/16/32 packed FP16 floating-point values in the source operand (the second operand) and stores the packed FP16 floating-point results in the destination operand.

The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$. For special cases, see Table 1-35.

The destination elements are updated according to the writemask.

Table 1-35. VRSQRTPH/VRSQRTSH Special Cases

Input value	Reset Value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	2^n	
X < 0	QNaN_Indefinite	Including −∞
X = −0	−∞	
X = +0	+∞	
X = +∞	+0	

Operation

VRSQRTPH dest{k1}, src

```
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := APPROXIMATE(1.0 / SQRT(tsrc) )
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRSQRTPH __m128h _mm_mask_rsqrt_ph (__m128h src, __mmask8 k, __m128h a);
VRSQRTPH __m128h _mm_maskz_rsqrt_ph (__mmask8 k, __m128h a);
VRSQRTPH __m128h _mm_rsqrt_ph (__m128h a);
VRSQRTPH __m256h _mm256_mask_rsqrt_ph (__m256h src, __mmask16 k, __m256h a);
VRSQRTPH __m256h _mm256_maskz_rsqrt_ph (__mmask16 k, __m256h a);
VRSQRTPH __m256h _mm256_rsqrt_ph (__m256h a);
VRSQRTPH __m512h _mm512_mask_rsqrt_ph (__m512h src, __mmask32 k, __m512h a);
VRSQRTPH __m512h _mm512_maskz_rsqrt_ph (__mmask32 k, __m512h a);
VRSQRTPH __m512h _mm512_rsqrt_ph (__m512h a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

VRSQRTSH—Compute Approximate Reciprocal of Square Root of Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 4F /r VRSQRTSH xmm1{k1}{z}, xmm2, xmm3/m16	A	V/V	AVX512-FP16 OR AVX10.1[1]	Compute the approximate reciprocal square root of the FP16 value in xmm3/m16 and store the result in the low word element of xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

NOTES:

[1] For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs the computation of the approximate reciprocal square-root of the low FP16 value in the second source operand (the third operand) and stores the result in the low word element of the destination operand (the first operand) according to the writemask k1.

The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL−1:128 of the destination operand are zeroed.

For special cases, see Table 1-35.

Operation

VRSQRTSH dest{k1}, src1, src2

```
VL = 128, 256 or 512
KL := VL/16

IF k1[0] or *no writemask*:
    DEST.fp16[0] := APPROXIMATE(1.0 / SQRT(src2.fp16[0]))
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
//else DEST.fp16[0] remains unchanged
DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRTSH __m128h _mm_mask_rsqrt_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VRSQRTSH __m128h _mm_maskz_rsqrt_sh (__mmask8 k, __m128h a, __m128h b);
VRSQRTSH __m128h _mm_rsqrt_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-58, “Type E10 Class Exception Conditions.”

VSCALEFPD—Scale Packed Float64 Values With Float64 Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Scale the packed double precision floating-point values in zmm2 using values from zmm3/m512/m64bcst. Under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a floating-point scale of the packed double precision floating-point values in the first source operand by multiplying them by 2 to the power of the double precision floating-point values in second source operand.

The equation of this operation is given by:

$\text{zmm1} := \text{zmm2}*2^{\text{floor(zmm3)}}$.

Floor(zmm3) means maximum integer value ? 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 1-36 and Table 1-37.

Table 1-36. VSCALEFPD/SD/PS/SS Special Cases

		Src2				Set IE
		±NaN	**+Inf**	**-Inf**	**0/Denorm/Norm**	
Src1	±QNaN	QNaN(Src1)	+INF	+0	QNaN(Src1)	IF either source is SNAN
	±SNaN	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	YES
	±Inf	QNaN(Src2)	Src1	QNaN_Indefinite	Src1	IF Src2 is SNAN or -INF
	±0	QNaN(Src2)	QNaN_Indefinite	Src1	Src1	IF Src2 is SNAN or +INF
	Denorm/Norm	QNaN(Src2)	±INF (Src1 sign)	±0 (Src1 sign)	Compute Result	IF Src2 is SNAN

Table 1-37. Additional VSCALEFPD/SD Special Cases

Special Case	Returned value	Faults
$\lvert result \rvert < 2^{-1074}$	±0 or ±Min-Denormal (Src1 sign)	Underflow
$\lvert result \rvert$? 2^{1024}	±INF (Src1 sign) or ±Max-normal (Src1 sign)	Overflow

Operation

```
SCALE(SRC1, SRC2)
{
TMP_SRC2 := SRC2
TMP_SRC1 := SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 64 bits floating-point value */
DEST[63:0] := TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
```

VSCALEFPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := SCALE(SRC1[i+63:i], SRC2[63:0]);
                ELSE DEST[i+63:i] := SCALE(SRC1[i+63:i], SRC2[i+63:i]);
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
```

```
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSCALEFPD __m512d _mm512_scalef_round_pd(__m512d a, __m512d b, int rounding);
VSCALEFPD __m512d _mm512_mask_scalef_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int rounding);
VSCALEFPD __m512d _mm512_maskz_scalef_round_pd(__mmask8 k, __m512d a, __m512d b, int rounding);
VSCALEFPD __m512d _mm512_scalef_pd(__m512d a, __m512d b);
VSCALEFPD __m512d _mm512_mask_scalef_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VSCALEFPD __m512d _mm512_maskz_scalef_pd(__mmask8 k, __m512d a, __m512d b);
VSCALEFPD __m256d _mm256_scalef_pd(__m256d a, __m256d b);
VSCALEFPD __m256d _mm256_mask_scalef_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VSCALEFPD __m256d _mm256_maskz_scalef_pd(__mmask8 k, __m256d a, __m256d b);
VSCALEFPD __m128d _mm_scalef_pd(__m128d a, __m128d b);
VSCALEFPD __m128d _mm_mask_scalef_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VSCALEFPD __m128d _mm_maskz_scalef_pd(__mmask8 k, __m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”

VSCALEFPH—Scale Packed FP16 Values with FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 2C /r VSCALEFPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Scale the packed FP16 values in xmm2 using values from xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 2C /r VSCALEFPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Scale the packed FP16 values in ymm2 using values from ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 2C /r VSCALEFPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Scale the packed FP16 values in zmm2 using values from zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a floating-point scale of the packed FP16 values in the first source operand by multiplying it by 2 to the power of the FP16 values in second source operand. The destination elements are updated according to the writemask.

The equation of this operation is given by:

$$zmm1 := zmm2 * 2^{floor(zmm3)}.$$

Floor(zmm3) means maximum integer value ? zmm3.

If the result cannot be represented in FP16, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand), is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits), and on the SAE bit.

Handling of special-case input values are listed in Table 1-38 and Table 1-39.

Table 1-38. VSCALEFPH/VSCALEFSH Special Cases

Src1	**Src2**				**Set IE**
	±NaN	**+INF**	**?INF**	**0/Denorm/Norm**	
±QNaN	QNaN(Src1)	+INF	+0	QNaN(Src1)	IF either source is SNaN
±SNaN	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	YES
±INF	QNaN(Src2)	Src1	QNaN_Indefinite	Src1	IF Src2 is SNaN or ?INF
±0	QNaN(Src2)	QNaN_Indefinite	Src1	Src1	IF Src2 is SNaN or +INF
Denorm/Norm	QNaN(Src2)	±INF (Src1 sign)	±0 (Src1 sign)	Compute Result	IF Src2 is SNaN

Table 1-39. Additional VSCALEFPH/VSCALEFSH Special Cases

Special Case	**Returned Value**	**Faults**
|result| < 2^{-24}	±0 or ±Min-Denormal (Src1 sign)	Underflow
|result| ? 2^{16}	±INF (Src1 sign) or ±Max-Denormal (Src1 sign)	Overflow

Operation

```
def scale_fp16(src1,src2):
    tmp1 := src1
    tmp2 := src2
    return tmp1 * POW(2, FLOOR(tmp2))
```

VSCALEFPH dest{k1}, src1, src2

```
VL = 128, 256, or 512
KL := VL / 16

IF (VL = 512) AND (EVEX.b = 1) and no memory operand:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and (EVEX.b = 1):
            tsrc := src2.fp16[0]
        ELSE:
            tsrc := src2.fp16[i]
        dest.fp16[i] := scale_fp16(src1.fp16[i],tsrc)
    ELSE IF *zeroing*:
        dest.fp16[i] := 0
    //else dest.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSCALEFPH __m128h _mm_mask_scalef_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSCALEFPH __m128h _mm_maskz_scalef_ph (__mmask8 k, __m128h a, __m128h b);
VSCALEFPH __m128h _mm_scalef_ph (__m128h a, __m128h b);
VSCALEFPH __m256h _mm256_mask_scalef_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VSCALEFPH __m256h _mm256_maskz_scalef_ph (__mmask16 k, __m256h a, __m256h b);
VSCALEFPH __m256h _mm256_scalef_ph (__m256h a, __m256h b);
VSCALEFPH __m512h _mm512_mask_scalef_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VSCALEFPH __m512h _mm512_maskz_scalef_ph (__mmask32 k, __m512h a, __m512h b);
VSCALEFPH __m512h _mm512_scalef_ph (__m512h a, __m512h b);
VSCALEFPH __m512h _mm512_mask_scalef_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, const int rounding);
VSCALEFPH __m512h _mm512_maskz_scalef_round_ph (__mmask32 k, __m512h a, __m512h b, const int;
VSCALEFPH __m512h _mm512_scalef_round_ph (__m512h a, __m512h b, const int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions”.

Denormal-operand exception (#D) is checked and signaled for src1 operand, but not for src2 operand. The denormal-operand exception is checked for src1 operand only if the src2 operand is not NaN. If the src2 operand is NaN, the processor generates NaN and does not signal denormal-operand exception, even if src1 operand is denormal.

VSCALEFPS—Scale Packed Float32 Values With Float32 Values

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Scale the packed single-precision floating-point values in zmm2 using floating-point values from zmm3/m512/m32bcst. Under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a floating-point scale of the packed single-precision floating-point values in the first source operand by multiplying them by 2 to the power of the float32 values in second source operand.

The equation of this operation is given by:

$zmm1 := zmm2*2^{floor(zmm3)}$.

Floor(zmm3) means maximum integer value ? zmm3.

If the result cannot be represented in single-precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is 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 1-36 and Table 1-40.

Table 1-40. Additional VSCALEFPS/SS Special Cases

Special Case	Returned value	Faults
$\lvert result \rvert < 2^{-149}$	±0 or ±Min-Denormal (Src1 sign)	Underflow
$\lvert result \rvert$? 2^{128}	±INF (Src1 sign) or ±Max-normal (Src1 sign)	Overflow

Operation

```
SCALE(SRC1, SRC2)
{                  ; Check for denormal operands
TMP_SRC2 := SRC2
TMP_SRC1 := SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 32 bits floating-point value */
DEST[31:0] := TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}
```

VSCALEFPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN DEST[i+31:i] := SCALE(SRC1[i+31:i], SRC2[31:0]);
                    ELSE DEST[i+31:i] := SCALE(SRC1[i+31:i], SRC2[i+31:i]);
                FI;
        ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                            ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSCALEFPS __m512 _mm512_scalef_round_ps(__m512 a, __m512 b, int rounding);
VSCALEFPS __m512 _mm512_mask_scalef_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int rounding);
VSCALEFPS __m512 _mm512_maskz_scalef_round_ps(__mmask16 k, __m512 a, __m512 b, int rounding);
VSCALEFPS __m512 _mm512_scalef_ps(__m512 a, __m512 b);
VSCALEFPS __m512 _mm512_mask_scalef_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VSCALEFPS __m512 _mm512_maskz_scalef_ps(__mmask16 k, __m512 a, __m512 b);
VSCALEFPS __m256 _mm256_scalef_ps(__m256 a, __m256 b);
VSCALEFPS __m256 _mm256_mask_scalef_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VSCALEFPS __m256 _mm256_maskz_scalef_ps(__mmask8 k, __m256 a, __m256 b);
VSCALEFPS __m128 _mm_scalef_ps(__m128 a, __m128 b);
VSCALEFPS __m128 _mm_mask_scalef_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VSCALEFPS __m128 _mm_maskz_scalef_ps(__mmask8 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”

VSCALEFSD—Scale Scalar Float64 Values With Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W1 2D /r VSCALEFSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	A	V/V	AVX512F OR AVX10.1[1]	Scale the scalar double precision floating-point values in xmm2 using the value from xmm3/m64. Under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a floating-point scale of the scalar double precision floating-point value in the first source operand by multiplying it by 2 to the power of the double precision floating-point value in second source operand.

The equation of this operation is given by:

$xmm1 := xmm2*2^{floor(xmm3)}$.

Floor(xmm3) means maximum integer value ≤ xmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 1-36 and Table 1-37.

Operation

```
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
TMP_SRC2 := SRC2
TMP_SRC1 := SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 64 bits floating-point value */
DEST[63:0] := TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
```

VSCALEFSD (EVEX encoded version)

```
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[63:0] := SCALE(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking*                        ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                    ; zeroing-masking
                DEST[63:0] := 0
        FI
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFSD __m128d _mm_scalef_round_sd(__m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_mask_scalef_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_maskz_scalef_round_sd(__mmask8 k, __m128d a, __m128d b, int);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions.”

VSCALEFSH—Scale Scalar FP16 Values with FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 2D /r VSCALEFSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Scale the FP16 values in xmm2 using the value from xmm3/m16 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a floating-point scale of the low FP16 element in the first source operand by multiplying it by 2 to the power of the low FP16 element in second source operand, storing the result in the low element of the destination operand.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

The equation of this operation is given by:

$$xmm1 := xmm2 * 2^{floor(xmm3)}.$$

Floor(xmm3) means maximum integer value ≤ xmm3.

If the result cannot be represented in FP16, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand), is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

Handling of special-case input values are listed in Table 1-38 and Table 1-39.

Operation

VSCALEFSH dest{k1}, src1, src2

```
IF (EVEX.b = 1) and no memory operand:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] or *no writemask*:
    dest.fp16[0] := scale_fp16(src1.fp16[0], src2.fp16[0]) // see VSCALEFPH
ELSE IF *zeroing*:
    dest.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFSH __m128h _mm_mask_scalef_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VSCALEFSH __m128h _mm_maskz_scalef_round_sh (__mmask8 k, __m128h a, __m128h b, const int rounding);
VSCALEFSH __m128h _mm_scalef_round_sh (__m128h a, __m128h b, const int rounding);
VSCALEFSH __m128h _mm_mask_scalef_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSCALEFSH __m128h _mm_maskz_scalef_sh (__mmask8 k, __m128h a, __m128h b);
VSCALEFSH __m128h _mm_scalef_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

Denormal-operand exception (#D) is checked and signaled for src1 operand, but not for src2 operand. The denormal-operand exception is checked for src1 operand only if the src2 operand is not NaN. If the src2 operand is NaN, the processor generates NaN and does not signal denormal-operand exception, even if src1 operand is denormal.

VSCALEFSS—Scale Scalar Float32 Value With Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 2D /r VSCALEFSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	A	V/V	AVX512F OR AVX10.1[1]	Scale the scalar single-precision floating-point value in xmm2 using floating-point value from xmm3/m32. Under writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a floating-point scale of the scalar single-precision floating-point value in the first source operand by multiplying it by 2 to the power of the float32 value in second source operand.

The equation of this operation is given by:

$xmm1 := xmm2*2^{floor(xmm3)}$.

Floor(xmm3) means maximum integer value ≤ xmm3.

If the result cannot be represented in single-precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 1-36 and Table 1-40.

Operation

```
SCALE(SRC1, SRC2)
{
                ; Check for denormal operands
TMP_SRC2 := SRC2
TMP_SRC1 := SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 32 bits floating-point value */
DEST[31:0] := TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}
```

VSCALEFSS (EVEX encoded version)

```
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[31:0] := SCALE(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] := 0
        FI
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSCALEFSS __m128 _mm_scalef_round_ss(__m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_mask_scalef_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_maskz_scalef_round_ss(__mmask8 k, __m128 a, __m128 b, int);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions.”

VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	Using signed qword indices, scatter double precision floating-point values to memory using writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

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)	N/A	N/A

Description

Stores up to 16 elements (or 8 elements) in doubleword/quadword vector zmm1 to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
- If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be scattered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be scattered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.

Operation

```
BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement
```

VSCATTERDPS (EVEX encoded versions)

```
(KL, VL)= (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[i+31:i]) * SCALE + DISP] :=
            SRC[i+31:i]
```

```
                k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VSCATTERDPD (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[k+31:k]) * SCALE + DISP] :=
            SRC[i+63:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VSCATTERQPS (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] :=
            SRC[i+31:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VSCATTERQPD (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP] :=
            SRC[i+63:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSCATTERDPD void _mm512_i32scatter_pd(void * base, __m256i vdx, __m512d a, int scale);
VSCATTERDPD void _mm512_mask_i32scatter_pd(void * base, __mmask8 k, __m256i vdx, __m512d a, int scale);
VSCATTERDPS void _mm512_i32scatter_ps(void * base, __m512i vdx, __m512 a, int scale);
VSCATTERDPS void _mm512_mask_i32scatter_ps(void * base, __mmask16 k, __m512i vdx, __m512 a, int scale);
VSCATTERQPD void _mm512_i64scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm512_mask_i64scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm512_i64scatter_ps(void * base, __m512i vdx, __m256 a, int scale);
VSCATTERQPS void _mm512_mask_i64scatter_ps(void * base, __mmask8 k, __m512i vdx, __m256 a, int scale);
VSCATTERDPD void _mm256_i32scatter_pd(void * base, __m128i vdx, __m256d a, int scale);
VSCATTERDPD void _mm256_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m256d a, int scale);
```

```
VSCATTERDPS void _mm256_i32scatter_ps(void * base, __m256i vdx, __m256 a, int scale);
VSCATTERDPS void _mm256_mask_i32scatter_ps(void * base, __mmask8 k, __m256i vdx, __m256 a, int scale);
VSCATTERQPD void _mm256_i64scatter_pd(void * base, __m256i vdx, __m256d a, int scale);
VSCATTERQPD void _mm256_mask_i64scatter_pd(void * base, __mmask8 k, __m256i vdx, __m256d a, int scale);
VSCATTERQPS void _mm256_i64scatter_ps(void * base, __m256i vdx, __m128 a, int scale);
VSCATTERQPS void _mm256_mask_i64scatter_ps(void * base, __mmask8 k, __m256i vdx, __m128 a, int scale);
VSCATTERDPD void _mm_i32scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERDPD void _mm_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERDPS void _mm_i32scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERDPS void _mm_mask_i32scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);
VSCATTERQPD void _mm_i64scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERQPD void _mm_mask_i64scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERQPS void _mm_i64scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERQPS void _mm_mask_i64scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);
```

SIMD Floating-Point Exceptions

Invalid, Overflow, Underflow, Precision, Denormal.

Other Exceptions

See Table 2-61, “Type E12 Class Exception Conditions.”

VSHUFF32x4/VSHUFF64x2/VSHUFI32x4/VSHUFI64x2—Shuffle Packed Values at 128-Bit Granularity

Opcode/ 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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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 AND AVX512F) OR AVX10.1[1]	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 OR AVX10.1[1]	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.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

256-bit Version: Moves one of the two 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 128-bit of the destination operand (first operand); moves one of the two packed 128-bit floating-point values from the second source operand (third operand) into the high 128-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

512-bit Version: Moves two of the four 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 256-bit of each double qword of the destination operand (first operand); moves two of the four packed 128-bit floating-point values from the second source operand (third operand) into the high 256-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

The first source operand is a vector register. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a vector register.

The writemask updates the destination operand with the granularity of 32/64-bit data elements.

Operation

```
Select2(SRC, control) {
CASE (control[0]) OF
    0:    TMP := SRC[127:0];
    1:    TMP := SRC[255:128];
ESAC;
RETURN TMP
}

Select4(SRC, control) {
CASE (control[1:0]) OF
    0:    TMP := SRC[127:0];
    1:    TMP := SRC[255:128];
    2:    TMP := SRC[383:256];
    3:    TMP := SRC[511:384];
ESAC;
RETURN TMP
}
```

VSHUFF32x4 (EVEX versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] := Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] := Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    THEN DEST[i+31:i] := 0
```

```
                FI;
        FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFF64x2 (EVEX 512-bit version)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] := Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] := Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFI32x4 (EVEX 512-bit version)

```
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] := Select4(SRC1[511:0], imm8[3:2]);
```

```
    TMP_DEST[383:256] := Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    THEN DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFI64x2 (EVEX 512-bit version)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] := Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] := Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFI32x4 __m512i _mm512_shuffle_i32x4(__m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i _mm512_mask_shuffle_i32x4(__m512i s, __mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i _mm512_maskz_shuffle_i32x4(__mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m256i _mm256_shuffle_i32x4(__m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i _mm256_mask_shuffle_i32x4(__m256i s, __mmask8 k, __m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i _mm256_maskz_shuffle_i32x4(__mmask8 k, __m256i a, __m256i b, int imm);
VSHUFF32x4 __m512 _mm512_shuffle_f32x4(__m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 _mm512_mask_shuffle_f32x4(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 _mm512_maskz_shuffle_f32x4(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFI64x2 __m512i _mm512_shuffle_i64x2(__m512i a, __m512i b, int imm);
VSHUFI64x2 __m512i _mm512_mask_shuffle_i64x2(__m512i s, __mmask8 k, __m512i b, __m512i b, int imm);
VSHUFI64x2 __m512i _mm512_maskz_shuffle_i64x2(__mmask8 k, __m512i a, __m512i b, int imm);
VSHUFF64x2 __m512d _mm512_shuffle_f64x2(__m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d _mm512_mask_shuffle_f64x2(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d _mm512_maskz_shuffle_f64x2(__mmask8 k, __m512d a, __m512d b, int imm);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-50, “Type E4NF Class Exception Conditions.”

Additionally:

#UD If EVEX.L’L = 0 for VSHUFF32x4/VSHUFF64x2.

VSQRTPH—Compute Square Root of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 51 /r VSQRTPH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Compute square roots of the packed FP16 values in xmm2/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 51 /r VSQRTPH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Compute square roots of the packed FP16 values in ymm2/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 51 /r VSQRTPH zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Compute square roots of the packed FP16 values in zmm2/m512/m16bcst, and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a packed FP16 square-root computation on the values from source operand and stores the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

Operation

```
VSQRTPH dest{k1}, src
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := SQRT(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTPH __m128h _mm_mask_sqrt_ph (__m128h src, __mmask8 k, __m128h a);
VSQRTPH __m128h _mm_maskz_sqrt_ph (__mmask8 k, __m128h a);
VSQRTPH __m128h _mm_sqrt_ph (__m128h a);
VSQRTPH __m256h _mm256_mask_sqrt_ph (__m256h src, __mmask16 k, __m256h a);
VSQRTPH __m256h _mm256_maskz_sqrt_ph (__mmask16 k, __m256h a);
VSQRTPH __m256h _mm256_sqrt_ph (__m256h a);
VSQRTPH __m512h _mm512_mask_sqrt_ph (__m512h src, __mmask32 k, __m512h a);
VSQRTPH __m512h _mm512_maskz_sqrt_ph (__mmask32 k, __m512h a);
VSQRTPH __m512h _mm512_sqrt_ph (__m512h a);
VSQRTPH __m512h _mm512_mask_sqrt_round_ph (__m512h src, __mmask32 k, __m512h a, const int rounding);
VSQRTPH __m512h _mm512_maskz_sqrt_round_ph (__mmask32 k, __m512h a, const int rounding);
VSQRTPH __m512h _mm512_sqrt_round_ph (__m512h a, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”

VSQRTSH—Compute Square Root of Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 51 /r VSQRTSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Compute square root of the low FP16 value in xmm3/m16 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a scalar FP16 square-root computation on the source operand and stores the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

```
VSQRTSH dest{k1}, src1, src2
IF k1[0] or *no writemask*:
    DEST.fp16[0] := SQRT(src2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTSH __m128h _mm_mask_sqrt_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VSQRTSH __m128h _mm_maskz_sqrt_round_sh (__mmask8 k, __m128h a, __m128h b, const int rounding);
VSQRTSH __m128h _mm_sqrt_round_sh (__m128h a, __m128h b, const int rounding);
VSQRTSH __m128h _mm_mask_sqrt_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSQRTSH __m128h _mm_maskz_sqrt_sh (__mmask8 k, __m128h a, __m128h b);
VSQRTSH __m128h _mm_sqrt_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VSUBPH—Subtract Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5C /r VSUBPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Subtract packed FP16 values from xmm3/m128/m16bcst to xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5C /r VSUBPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Subtract packed FP16 values from ymm3/m256/m16bcst to ymm2, and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5C /r VSUBPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Subtract packed FP16 values from zmm3/m512/m16bcst to zmm2, and store the result in zmm1 subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction subtracts packed FP16 values from second source operand from the corresponding elements in the first source operand, storing the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

Operation

VSUBPH (EVEX Encoded Versions) When SRC2 Operand is a Register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.fp16[j] := SRC1.fp16[j] - SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VSUBPH (EVEX Encoded Versions) When SRC2 Operand is a Memory Source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            DEST.fp16[j] := SRC1.fp16[j] - SRC2.fp16[0]
        ELSE:
            DEST.fp16[j] := SRC1.fp16[j] - SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSUBPH __m128h _mm_mask_sub_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSUBPH __m128h _mm_maskz_sub_ph (__mmask8 k, __m128h a, __m128h b);
VSUBPH __m128h _mm_sub_ph (__m128h a, __m128h b);
VSUBPH __m256h _mm256_mask_sub_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VSUBPH __m256h _mm256_maskz_sub_ph (__mmask16 k, __m256h a, __m256h b);
VSUBPH __m256h _mm256_sub_ph (__m256h a, __m256h b);
VSUBPH __m512h _mm512_mask_sub_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VSUBPH __m512h _mm512_maskz_sub_ph (__mmask32 k, __m512h a, __m512h b);
VSUBPH __m512h _mm512_sub_ph (__m512h a, __m512h b);
VSUBPH __m512h _mm512_mask_sub_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VSUBPH __m512h _mm512_maskz_sub_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);
VSUBPH __m512h _mm512_sub_round_ph (__m512h a, __m512h b, int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”

VSUBSH—Subtract Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 5C /r VSUBSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Subtract the low FP16 value in xmm3/m16 from xmm2 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction subtracts the low FP16 value from the second source operand from the corresponding value in the first source operand, storing the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VSUBSH (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC2 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := SRC1.fp16[0] - SRC2.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSUBSH __m128h _mm_mask_sub_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
VSUBSH __m128h _mm_maskz_sub_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
VSUBSH __m128h _mm_sub_round_sh (__m128h a, __m128h b, int rounding);
VSUBSH __m128h _mm_mask_sub_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSUBSH __m128h _mm_maskz_sub_sh (__mmask8 k, __m128h a, __m128h b);
VSUBSH __m128h _mm_sub_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

VUCOMISH—Unordered Compare Scalar FP16 Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.MAP5.W0 2E /r VUCOMISH xmm1, xmm2/m16 {sae}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Compare low FP16 values in xmm1 and xmm2/m16 and set the EFLAGS flags accordingly.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction compares the FP16 values in the low word of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 16-bit memory location.

The VUCOMISH instruction differs from the VCOMISH instruction in that it signals a SIMD floating-point invalid operation exception (#I) only if a source operand is an SNaN. The COMISS instruction signals an invalid numeric exception when a source operand is either a QNaN or SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Operation

VUCOMISH

```
RESULT := UnorderedCompare(SRC1.fp16[0],SRC2.fp16[0])
if RESULT is UNORDERED:
    ZF, PF, CF := 1, 1, 1
else if RESULT is GREATER_THAN:
    ZF, PF, CF := 0, 0, 0
else if RESULT is LESS_THAN:
    ZF, PF, CF := 0, 0, 1
else: // RESULT is EQUALS
    ZF, PF, CF := 1, 0, 0

OF, AF, SF := 0, 0, 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VUCOMISH int _mm_ucomieq_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomige_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomigt_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomile_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomilt_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomineq_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”

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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	Return the bitwise logical XOR of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

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 Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E4 Class Exception Conditions.”

XORPS—Bitwise Logical XOR of Packed Single Precision Floating-Point Values

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 AND AVX512DQ) OR AVX10.1[1]	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 AND AVX512DQ) OR AVX10.1[1]	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 OR AVX10.1[1]	Return the bitwise logical XOR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

NOTES:

1. For instructions with a CPUID feature flag specifying AVX10, the programmer must check the available vector options on the processor at run-time via CPUID Leaf 24H, the Intel AVX10 Converged Vector ISA Leaf. This leaf enumerates the maximum supported vector width and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

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 Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E4 Class Exception Conditions.”