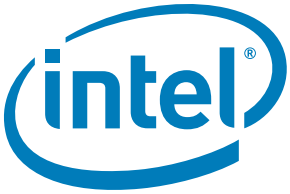


Intel® Architecture Instruction Set Extensions Programming Reference

319433-015

JULY 2013

CHAPTER 1
FUTURE INTEL® ARCHITECTURE INSTRUCTION EXTENSIONS

CHAPTER 2
INTEL® AVX-512 APPLICATION PROGRAMMING MODEL

CHAPTER 3
SYSTEM PROGRAMMING FOR INTEL® AVX-512

CHAPTER 4
AVX-512 INSTRUCTION ENCODING

CHAPTER 5
INSTRUCTION SET REFERENCE, A-Z

CHAPTER 6
INSTRUCTION SET REFERENCE - OPMASK

CHAPTER 7
ADDITIONAL 512-BIT INSTRUCTION EXTENSIONS

CHAPTER 8
INTEL® SHA EXTENSIONS

CHAPTER 9
INTEL® MEMORY PROTECTION EXTENSIONS

CHAPTER 10
ADDITIONAL NEW INSTRUCTIONS

CHAPTER 11
INTEL® PROCESSOR TRACE

TABLES

FIGURES

PAGE

CHAPTER 1
FUTURE INTEL® ARCHITECTURE INSTRUCTION EXTENSIONS

1.1 ABOUT THIS DOCUMENT

This document describes the software programming interfaces of Intel® architecture instruction extensions for future Intel processor generations. The instruction set extensions cover a diverse range of application domains and programming usages. There are 512-bit SIMD vector instruction extensions, instruction set extensions targeting memory protection issues such as buffer overruns, and extensions targeting secure hash algorithm (SHA) accelerations like SHA1 and SHA256.

The 512-bit SIMD vector SIMD extensions, referred to as Intel® AVX-512 instructions, deliver comprehensive set of functionality and higher performance than AVX and AVX2 family of instructions. AVX and AVX2 are covered in *Intel® 64 and IA-32 Architectures Software Developer's Manual sets.* The reader can refer to them for basic and more background information related to various features referenced in this document.

The base of the 512-bit SIMD instruction extensions are referred to as Intel® AVX-512 Foundation instructions. They include extensions of the AVX and AVX2 family of SIMD instructions but are encoded using a new encoding scheme with support for 512-bit vector registers, up to 32 vector registers in 64-bit mode, and conditional processing using opmask registers.

Chapters 2 through 6 are devoted to the programming interfaces of the AVX-512 Foundation instruction set.

Chapter 7 covers additional 512-bit SIMD instruction extensions that targets specific application domain, AVX-512 Conflict Detection instructions can speed up histogram operations, AVX-512 Exponential and Reciprocal instructions for certain transcendental mathematical computations, and AVX-512 Prefetch instructions for specific prefetch operations.

Chapter 8 covers instruction set extensions targeted for SHA acceleration. Chapter 9 describes instruction set extensions that offer software tools with capability to address memory protection issues such as buffer overruns.

For an overview and detailed descriptions of hardware -accelerated SHA extensions, and Intel® Memory Protection Extensions (Intel® MPX), see the respective chapters.

Chapter 10 covers instructions operating on general purpose registers in future Intel processors. Chapter 11 describes the architecture of Intel® Processor Trace, which allows software to capture data packets with low overhead and to reconstruct detailed control flow information of program execution.

1.2 INTEL® AVX-512 INSTRUCTIONS ARCHITECTURE OVERVIEW

Intel AVX-512 Foundation instructions are a natural extension to AVX and AVX2. It introduces the following architectural enhancements:

- Support for 512-bit wide vectors and SIMD register set. 512-bit register state is managed by the operating system using XSAVE/XRSTOR instructions introduced in 45 nm Intel 64 processors (see *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B,* and *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).
- Support for 16 new, 512-bit SIMD registers (for a total of 32 SIMD registers, ZMM0 through ZMM31) in 64-bit mode. The extra 16 registers state is managed by the operating system using XSAVE/XRSTOR/XSAVEOPT.
- Support for 8 new opmask registers (k0 through k7) used for conditional execution and efficient merging of destination operands. Again, the opmask register state is managed by the operating system using XSAVE/XRSTOR/XSAVEOPT instructions
- A new encoding prefix (referred to as EVEX) to support additional vector length encoding up to 512 bits. The EVEX prefix builds upon the foundations of VEX prefix, to provide compact, efficient encoding for functionality available to VEX encoding plus the following enhanced vector capabilities:
    - opmasks
    - embedded broadcast
    - instruction prefix-embedded rounding control

- compressed address displacements

1.2.1 512-Bit Wide SIMD Register Support

AVX-512 instructions support 512-bit wide SIMD registers (ZMM0-ZMM31). The lower 256-bits of the ZMM registers are aliased to the respective 256-bit YMM registers and the lower 128-bit are aliased to the respective 128-bit XMM registers.

1.2.2 32 SIMD Register Support

AVX-512 instructions also support for 32 SIMD registers in 64-bit mode (XMM0-XMM31, YMM0-YMM31 and ZMM0-ZMM31). The number of available vector registers in 32-bit mode is still 8.

1.2.3 Eight Opmask Register Support

AVX-512 instructions support 8 opmask registers (k0-k7). The width of each opmask register is architecturally defined of size MAX_KL (64 bits). Seven of the eight opmask registers (k1-k7) can be used in conjunction with EVEX-encoded AVX-512 Foundation instructions to provide conditional execution and efficient merging of data elements in the destination operand. The encoding of opmask register k0 is typically used when all data elements (unconditional processing) are desired. Additionally, the opmask registers are also used as vector flags/element-level vector sources to introduce novel SIMD functionality as seen in new instructions such as VCOMPRESSPS.

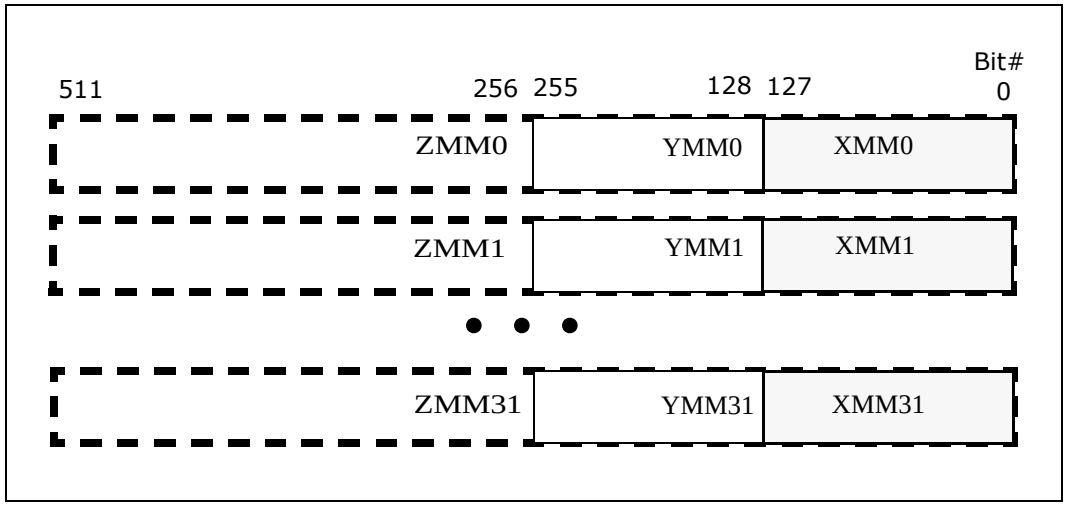


Figure 1-1. 512-Bit Wide Vectors and SIMD Register Set

1.2.4 Instruction Syntax Enhancement

The architecture of EVEX encoding enhances vector instruction encoding scheme in the following way:

- 512-bit vector-length, up to 32 ZMM registers, and enhanced vector programming environment are supported using the enhanced VEX (EVEX).

The EVEX prefix provides more encodable bit fields than VEX prefix. In addition to encoding 32 ZMM registers in 64-bit mode, instruction encoding using the EVEX can directly encode 7 (out of 8) opmask register operands to provide conditional processing in vector instruction programming. The enhanced vector programming environment can be explicitly expressed in the instruction syntax to include the following elements:

- An opmask operand: the opmask registers are expressed using the notation “k1” through “k7”. An EVEX-encoded instruction supporting conditional vector operation using the opmask register k1 is expressed by attaching the notation {k1} next to the destination operand. The use of this feature is optional for most instruc-

tions. There are two types of masking (merging and zeroing) differentiated using the EVEX.z bit ({z} in instruction signature).

- Embedded broadcast may be supported for some instructions on the source operand that can be encoded as a memory vector. Since the data elements of a memory vector may be conditionally fetched or written to, the memory operand is expressed using a variable length notation "mv".
- For instruction syntax that operates only on floating-point data in SIMD registers with rounding semantics, the EVEX can provide explicit rounding control within the EVEX bit fields at either scalar or 512-bit vector length.

In AVX-512 instructions, vector addition of all elements of the source operands can be expressed in the same syntax as AVX instruction:

```
VADDPS zmm1, zmm2, zmm3
```

Additionally, the EVEX encoding scheme of AVX-512 Foundation can express conditional vector addition as

```
VADDPS zmm1 {k1}{z}, zmm2, zmm3
```

where

- conditional processing and updates to destination is expressed with an opmask register,
- zeroing behavior of the opmask selected destination element is expressed by the {z} modifier (with merging as the default if no modifier specified),

Note that some SIMD instructions supporting three-operand syntax but processing only less or equal than 128-bits of data are considered part of the 512-bit SIMD instruction set extensions, because bits MAX_VL-1:128 of the destination register are zeroed by the processor. The same rule applies to instructions operating on 256-bits of data where bits MAX_VL-1:256 of the destination register are zeroed.

1.2.5 EVEX Instruction Encoding Support

Intel AVX-512 instructions employ a new encoding prefix, referred to as EVEX, in the Intel 64 and IA-32 instruction encoding format. Instruction encoding using the EVEX prefix provides the following capabilities:

- Direct encoding of a SIMD register operand within EVEX (similar to VEX). This provides instruction syntax support for three source operands.
- Compaction of REX prefix functionality and extended SIMD register encoding: The equivalent REX-prefix compaction functionality offered by the VEX prefix is provided within EVEX. Furthermore, EVEX extends the operand encoding capability to allow direct addressing of up to 32 ZMM registers in 64-bit mode.
- Compaction of SIMD prefix functionality and escape byte encoding: The functionality of SIMD prefix (66H, F2H, F3H) on opcode is equivalent to an opcode extension field to introduce new processing primitives. This functionality is provided in the VEX prefix encoding scheme and employed within the EVEX prefix. Similarly, the functionality of the escape opcode byte (0FH) and two-byte escape (0F38H, 0F3AH) are also compacted within the EVEX prefix encoding.
- Most EVEX-encoded SIMD numeric and data processing instruction semantics with memory operand have relaxed memory alignment requirements than instructions encoded using SIMD prefixes (see Section 2.4, "Memory Alignment").
- Direct encoding of a opmask operand within the EVEX prefix. This provides instruction syntax support for conditional vector-element operation and merging of destination operand using an opmask register (k1-k7).
- Direct encoding of a broadcast attribute for instructions with a memory operand source. This provides instruction syntax support for elements broadcasting of the second operand before being used in the actual operation.
- Compressed memory address displacements for a more compact instruction encoding byte sequence.

EVEX encoding applies to SIMD instructions operating on XMM, YMM and ZMM registers. EVEX is not supported for instructions operating on MMX or x87 registers. Details of EVEX instruction encoding are discussed in Chapter 4.

This page was intentionally left blank.

CHAPTER 2
INTEL® AVX-512 APPLICATION PROGRAMMING MODEL

The application programming model for AVX-512 Foundation instructions, described in Chapter 5, and other 512-bit instructions, described in Chapter 7, extend from that of AVX and AVX2 with differences detailed in this chapter.

2.1 DETECTION OF AVX-512 FOUNDATION INSTRUCTIONS

The majority of AVX-512 Foundation instructions are encoded using the EVEX encoding scheme. EVEX-encoded instructions can operate on the 512-bit ZMM register state plus 8 opmask registers. The opmask instructions in AVX-512 Foundation instructions operate only on opmask registers or with a general purpose register. System software requirements to support ZMM state and opmask instructions are described in Chapter 3, "System Programming For Intel® AVX-512".

Processor support of AVX-512 Foundation instructions is indicated by CPUID.(EAX=07H, ECX=0):EBX.AVX512F[bit 16] = 1. Detection of AVX-512 Foundation instructions operating on ZMM states and opmask registers need to follow the general procedural flow in Figure 2-1.

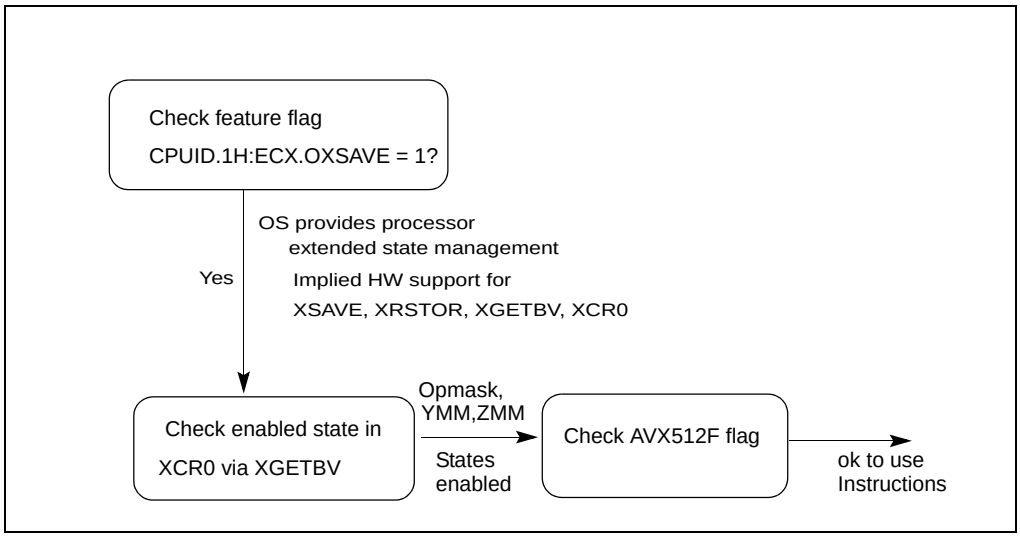


Figure 2-1. Procedural Flow of Application Detection of AVX-512 Foundation Instructions

Prior to using AVX-512 Foundation instructions, the application must identify that the operating system supports the XGETBV instruction, the ZMM register state, in addition to processor's support for ZMM state management using XSAVE/XRSTOR and AVX-512 Foundation instructions. The following simplified sequence accomplishes both and is strongly recommended.

1) Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use[1])

2) Execute XGETBV and verify that XCR0[7:5] = '111b' (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = '11b' (XMM state and YMM state are enabled by OS).

3) Detect CPUID.0x7.0:EBX.AVX512F[bit 16] = 1.

1. If CPUID.01H:ECX.OSXSAVE reports 1, it also indirectly implies the processor supports XSAVE, XRSTOR, XGETBV, processor extended state bit vector XCR0 register. Thus an application may streamline the checking of CPUID feature flags for XSAVE and OSXSAVE. XSETBV is a privileged instruction.

2.2 ACCESSING XMM, YMM AND ZMM REGISTERS

The lower 128 bits of a YMM register is aliased to the corresponding XMM register. Legacy SSE instructions (i.e. SIMD instructions operating on XMM state but not using the VEX prefix, also referred to non-VEX encoded SIMD instructions) will not access the upper bits (MAX_VL-1:128) of the YMM registers. AVX and FMA instructions with a VEX prefix and vector length of 128-bits zeroes the upper 128 bits of the YMM register.

Upper bits of YMM registers (255:128) can be read and written by many instructions with a VEX.256 prefix.

XSAVE and XRSTOR may be used to save and restore the upper bits of the YMM registers.

The lower 256 bits of a ZMM register are aliased to the corresponding YMM register. Legacy SSE instructions (i.e. SIMD instructions operating on XMM state but not using the VEX prefix, also referred to non-VEX encoded SIMD instructions) will not access the upper bits (MAX_VL-1:128) of the ZMM registers, where MAX_VL is maximum vector length (currently 512 bits). AVX and FMA instructions with a VEX prefix and vector length of 128-bits zero the upper 384 bits of the ZMM register, while VEX prefix and vector length of 256-bits zeros the upper 256 bits of the ZMM register.

Upper bits of ZMM registers (511:256) can be read and written by instructions with an EVEX.512 prefix.

2.3 ENHANCED VECTOR PROGRAMMING ENVIRONMENT USING EVEX ENCODING

EVEX-encoded AVX-512 instructions support an enhanced vector programming environment. The enhanced vector programming environment uses the combination of EVEX bit-field encodings and a set of eight opmask registers to provide the following capabilities:

- Conditional vector processing of EVEX-encoded instruction. Opmask registers k1 through k7 can be used to conditionally govern the per-data-element computational operation and the per-element updates to the destination operand of an AVX-512 Foundation instruction. Each bit of the opmask register governs one vector element operation (a vector element can be of 32 bits or 64 bits).
- In addition to providing predication control on vector instructions via EVEX bit-field encoding, the opmask registers can also be used similarly to general-purpose registers as source/destination operands using modR/M encoding for non-mask-related instructions. In this case, an opmask register k0 through k7 can be selected.
- In 64-bit mode, 32 vector registers can be encoded using EVEX prefix.
- Broadcast may be supported for some instructions on the operand that can be encoded as a memory vector. Since the data elements of a memory vector may be conditionally fetched or written to, and the vector size is dependent on the data transformation function, the memory operand is expressed using a variable length notation “mv”.
- Flexible rounding control for register-to-register flavor of EVEX encoded 512-bit and scalar instructions. Four rounding modes are supported by direct encoding within the EVEX prefix overriding MXCSR settings.
- Broadcast of one element to the rest of the destination vector register.
- Compressed 8-bit displacement encoding scheme to increase the instruction encoding density for instructions that normally require disp32 syntax.

2.3.1 OPMASK Register to Predicate Vector Data Processing

AVX-512 instructions using EVEX encodes a predicate operand to conditionally control per-element computational operation and updating of result to the destination operand. The predicate operand is known as the opmask register. The opmask is a set of eight architectural registers of size MAX_KL (64-bit). Note that from this set of 8 architectural registers, only k1 through k7 can be addressed as predicate operand. k0 can be used as a regular source or destination but cannot be encoded as a predicate operand. Note also that a predicate operand can be used to enable memory fault-suppression for some instructions with a memory operand (source or destination).

As a predicate operand, the opmask registers contain one bit to govern the operation/update to each data element of a vector register. In general, opmask registers can support instructions with element sizes: single-precision floating-point (float32), integer doubleword(int32), double-precision floating-point (float64), integer quadword (int64). The length of a opmask register, MAX_KL, is sufficient to handle up to 64 elements with one bit per

element, i.e. 64 bits. Masking is supported in most of the AVX-512 instructions. For a given vector length, each instruction accesses only the number of least significant mask bits that are needed based on its data type. For example, AVX-512 Foundation instructions operating on 64-bit data elements with a 512-bit vector length, only use the 8 least significant bits of the opmask register.

An opmask register affects an AVX-512 instruction at per-element granularity. So, any numeric or non-numeric operation of each data element and per-element updates of intermediate results to the destination operand are predicated on the corresponding bit of the opmask register.

An opmask serving as a predicate operand in AVX-512 obeys the following properties:

- The instruction's operation is not performed for an element if the corresponding opmask bit is not set. This implies that no exception or violation can be caused by an operation on a masked-off element. Consequently, no MXCSR exception flag is updated as a result of a masked-off operation.
- A destination element is not updated with the result of the operation if the corresponding writemask bit is not set. Instead, the destination element value must be preserved (merging-masking) or it must be zeroed out (zeroing-masking).
- For some instructions with a memory operand, memory faults are suppressed for elements with a mask bit of 0.

Note that this feature provides a versatile construct to implement control-flow predication as the mask in effect provides a merging behavior for AVX-512 vector register destinations. As an alternative the masking can be used for zeroing instead of merging, so that the masked out elements are updated with 0 instead of preserving the old value. The zeroing behavior is provided to remove the implicit dependency on the old value when it is not needed.

Most instructions with masking enabled accept both forms of masking. Instructions that must have EVEX.aaa bits different than 0 (gather and scatter) and instructions that write to memory only accept merging-masking.

It's important to note that the per-element destination update rule also applies when the destination operand is a memory location. Vectors are written on a per element basis, based on the opmask register used as a predicate operand.

The value of an opmask register can be:

- generated as a result of a vector instruction (e.g. CMP)
- loaded from memory
- loaded from GPR register
- or modified by mask-to-mask operations

Opmask registers can be used for purposes outside of predication. For example, they can be used to manipulate sparse sets of elements from a vector or used to set the EFLAGS based on the 0/0xFFFFFFFFFFFFFFFF/other status of the OR of two opmask registers.

2.3.1.1 Opmask Register K0

The only exception to the opmask rules described above is that opmask k0 can not be used as a predicate operand. Opmask k0 cannot be encoded as a predicate operand for a vector operation; the encoding value that would select opmask k0 will instead selects an implicit opmask value of 0xFFFFFFFFFFFFFFFF, thereby effectively disabling masking. Opmask register k0 can still be used for any instruction that takes opmask register(s) as operand(s) (either source or destination).

Note that certain instructions implicitly use the opmask as an extra destination operand. In such cases, trying to use the "no mask" feature will translate into a #UD fault being raised.

2.3.1.2 Example of Opmask Usages

The example below illustrates predicated vector add operation and predicated updates of added results into the destination operand. The initial state of vector registers zmm0, zmm1, and zmm2 and k3 are:

```
MSB.....................................LSB

zmm0 =
[ 0x00000003 0x00000002 0x00000001 0x00000000 ] (bytes 15 through 0)
[ 0x00000007 0x00000006 0x00000005 0x00000004 ] (bytes 31 through 16)
```

```
[ 0x0000000B 0x0000000A 0x00000009 0x00000008 ] (bytes 47 through 32)
[ 0x0000000F 0x0000000E 0x0000000D 0x0000000C ] (bytes 63 through 48)

zmm1 =
[ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ] (bytes 15 through 0)
[ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ] (bytes 31 through 16)
[ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ] (bytes 47 through 32)
[ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ] (bytes 63 through 48)

zmm2 =
[ 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA ] (bytes 15 through 0)
[ 0xBBBBBBBB 0xBBBBBBBB 0xBBBBBBBB 0xBBBBBBBB ] (bytes 31 through 16)
[ 0xCCCCCCCC 0xCCCCCCCC 0xCCCCCCCC 0xCCCCCCCC ] (bytes 47 through 32)
[ 0xDDDDDDDD 0xDDDDDDDD 0xDDDDDDDD 0xDDDDDDDD ] (bytes 63 through 48)

k3 = 0x8F03 (1000 1111 0000 0011)
```

An opmask register serving as a predicate operand is expressed as a curly-braces-enclosed decorator following the first operand in the Intel assembly syntax. Given this state, we will execute the following instruction:

```
vpaddd zmm2 {k3}, zmm0, zmm1
```

The vpaddd instruction performs 32-bit integer additions on each data element conditionally based on the corresponding bit value in the predicate operand k3. Since per-element operations are not operated if the corresponding bit of the predicate mask is not set, the intermediate result is:

```
[ ********** ********** 0x00000010 0x0000000F ] (bytes 15 through 0)
[ ********** ********** ********** ********** ] (bytes 31 through 16)
[ 0x0000001A 0x00000019 0x00000018 0x00000017 ] (bytes 47 through 32)
[ 0x0000001E ********** ********** ********** ] (bytes 63 through 48)
```

where ”**********” indicates that no operation is performed.

This intermediate result is then written into the destination vector register, zmm2, using the opmask register k3 as the writemask, producing the following final result:

```
zmm2 =
[ 0xAAAAAAAA 0xAAAAAAAA 0x00000010 0x0000000F ] (bytes 15 through 0)
[ 0xBBBBBBBB 0xBBBBBBBB 0xBBBBBBBB 0xBBBBBBBB ] (bytes 31 through 16)
[ 0x0000001A 0x00000019 0x00000018 0x00000017 ] (bytes 47 through 32)
[ 0x0000001E 0xDDDDDDDD 0xDDDDDDDD 0xDDDDDDDD ] (bytes 63 through 48)
```

Note that for a 64-bit instruction (say vaddpd), only the 8 LSB of mask k3 (0x03) would be used to identify the predicate operation on each one of the 8 elements of the source/destination vectors.

2.3.2 OpMask Instructions

AVX-512 Foundation instructions provide a collection of opmask instructions that allow programmers to set, copy, or operate on the contents of a given opmask register. There are three types of opmask instructions:

- **Mask read / write instructions**: These instructions move data between a general-purpose integer register or memory and an opmask mask register, or between two opmask registers. For example:
  - kmovw k1, ebx; move lower 16 bits of ebx to k1.
- **Flag instructions**: This category, consisting of instructions that modify EFLAGS based on the content of opmask registers.
  - kortestw k1, k2; OR registers k1 and k2 and updated EFLAGS accordingly.
- **Mask logical instructions**: These instructions perform standard bitwise logical operations between opmask registers.
  - kandw k1, k2, k3; AND lowest 16 bits of registers k2 and k3, leaving the result in k1.

2.3.3 Broadcast

EVEX encoding provides a bit-field to encode data broadcast for some load-op instructions, i.e. instructions that load data from memory and perform some computational or data movement operation. A source element from memory can be broadcasted (repeated) across all the elements of the effective source operand (up to 16 times for 32-bit data element, up to 8 times for 64-bit data element). The is useful when we want to reuse the same scalar operand for all the operations in a vector instruction. Broadcast is only enabled on instructions with an element size of 32 bits or 64 bits. Byte and word instructions do not support embedded broadcast.
The functionality of data broadcast is expressed as a curly-braces-enclosed decorator following the last register/memory operand in the Intel assembly syntax.

For instance:
vmulps zmm1, zmm2, [rax] {1to16}

The {1to16} primitive loads one float32 (single precision) element from memory, replicates it 16 times to form a vector of 16 32-bit floating-point elements, multiplies the 16 float32 elements with the corresponding elements in the first source operand vector, and put each of the 16 results into the destination operand.

AVX-512 instructions with store semantics and pure load instructions do not support broadcast primitives.

```
vmovaps [rax] {k3}, zmm19
```

In contrast, the k3 opmask register is used as the predicate operand in the above example. Only the store operation on data elements corresponding to the non-zero bits in k3 will be performed.

2.3.4 STATIC ROUNDING MODE AND SUPPRESS ALL EXCEPTIONS

In previous SIMD instruction extensions, rounding control is generally specified in MXCSR, with a handful of instructions providing per-instruction rounding override via encoding fields within the imm8 operand. AVX-512 offers a more flexible encoding attribute to override MXCSR-based rounding control for floating-pointing instruction with rounding semantic. This rounding attribute embedded in the EVEX prefix is called Static (per instruction) Rounding Mode or Rounding Mode override. This attribute allows programmers to statically apply a specific arithmetic rounding mode irrespective of the value of RM bits in MXCSR. It is available only to register-to-register flavors of EVEX-encoded floating-point instructions with rounding semantic. The differences between these three rounding control interfaces are summarized in Table 2-1.

Table 2-1. Characteristics of Three Rounding Control Interfaces

Rounding Interface	Static Rounding Override	Imm8 Embedded Rounding Override	MXCSR Rounding Control
Semantic Requirement	FP rounding	FP rounding	FP rounding
Prefix Requirement	EVEX.B = 1	NA	NA
Rounding Control	EVEX.L'L	IMM8[1:0] or MXCSR.RC (depending on IMM8[2])	MXCSR.RC
Suppress All Exceptions (SAE)	Implied	no	no
SIMD FP Exception #XF	All suppressed	Can raise #I, #P (unless SPE is set)	MXCSR masking controls
MXCSR flag update	No	yes (except PE if SPE is set)	Yes
Precedence	Above MXCSR.RC	Above EVEX.L'L	Default
Scope	512-bit, reg-reg, Scalar reg-reg	ROUNDPx, ROUNDSx, VCVTPS2PH, VRNDSCALExx	All SIMD operands, vector lengths

The static rounding-mode override in AVX-512 also implies the "suppress-all-exceptions" (SAE) attribute. The SAE effect is as if all the MXCSR mask bits are set, and none of the MXCSR flags will be updated. Using static rounding-mode via EVEX without SAE is not supported.

Static Rounding Mode and SAE control can be enabled in the encoding of the instruction by setting the EVEX.b bit to 1 in a register-register vector instruction. In such a case, vector length is assumed to be MAX_VL (512-bit in case of AVX-512 packed vector instructions). Table 2-2 summarizes the possible static rounding-mode assignments in AVX-512 instructions.

Note that some instructions already allow to specify the rounding mode statically via immediate bits. In such case, the immediate bits take precedence over the embedded rounding mode (in the same vein that they take precedence over whatever MXCSR.RM says).

Table 2-2. Static Rounding Mode

Function	Description
{rn-sae}	Round to nearest (even) + SAE
{rd-sae}	Round down (toward -inf) + SAE
{ru-sae}	Round up (toward +inf) + SAE
{rz-sae}	Round toward zero (Truncate) + SAE

An example of use would be in the following instructions:

```
vaddps zmm7 {k6}, zmm2, zmm4, {rd-sae}
```

Which would perform the single-precision floating-point addition of vectors zmm2 and zmm4 with round-towards-minus-infinity, leaving the result in vector zmm7 using k6 as conditional writemask.

Note that MXCSR.RM bits are ignored and unaffected by the outcome of this instruction.

Examples of instructions instances where the static rounding-mode is not allowed would be:

```
; rounding-mode already specified in the instruction immediate
vrndscaleps zmm7 {k6}, zmm2, 0x00
```

```
; instructions with memory operands
vmulps zmm7 {k6}, zmm2,[rax], {rd-sae}
```

2.3.5 Compressed Disp8*N Encoding

EVEX encoding supports a new displacement representation that allows for a more compact encoding of memory addressing commonly used in unrolled code, where an 8-bit displacement can address a range exceeding the dynamic range of an 8-bit value. This compressed displacement encoding is referred to as disp8*N, where N is a constant implied by the memory operation characteristic of each instruction.

The compressed displacement is based on the assumption that the effective displacement (of a memory operand occurring in a loop) is a multiple of the granularity of the memory access of each iteration. Since the Base register in memory addressing already provides byte-granular resolution, the lower bits of the traditional disp8 operand becomes redundant, and can be implied from the memory operation characteristic.

The memory operation characteristics depend on the following:

- The destination operand is updated as a full vector, a single element, or multi-element tuples.
- The memory source operand (or vector source operand if the destination operand is memory) is fetched (or treated) as a full vector, a single element, or multi-element tuples.

For example,

```
vaddps zmm7, zmm2, disp8[membase + index*8]
```

The destination zmm7 is updated as a full 512-bit vector, and 64-bytes of data are fetched from memory as a full vector; the next unrolled iteration may fetch from memory in 64-byte granularity per iteration. There are 6 bits of lowest address that can be compressed, hence N = 2^6 = 64. The contribution of "disp8" to effective address calculation is 64*disp8.

```
vbroadcastf32x4 zmm7, disp8[membase + index*8]
```

In VBROADCASTF32x4, memory is fetched as a 4tuple of 4 32-bit entities. Hence the common lowest address bits that can be compressed is 4, corresponding to the 4tuple width of 2^4 = 16 bytes (4x32 bits).Therefore, N = 2^4.

For EVEX encoded instructions that update only one element in the destination, or source element is fetched individually, the number of lowest address bits that can be compressed is generally the width in bytes of the data element, hence N = 2^(width).

2.4 MEMORY ALIGNMENT

Memory alignment requirements on EVEX-encoded SIMD instructions are similar to VEX-encoded SIMD instructions. Memory alignment applies to EVEX-encoded SIMD instructions in three categories:

- Explicitly-aligned SIMD load and store instructions accessing 64 bytes of memory with EVEX prefix encoded vector length of 512 bits(e.g. VMOVAPD, VMOVAPS, VMOVDQA, etc.). These instructions always require memory address to be aligned on 64-byte boundary.
- Explicitly-unaligned SIMD load and store instructions accessing 64 bytes or less of data from memory (e.g. VMOVUPD, VMOVUPS, VMOVDQU, VMOVQ, VMOVD, etc.). These instructions do not require memory address to be aligned on natural vector-length byte boundary.
- Most arithmetic and data processing instructions encoded using EVEX support memory access semantics. When these instructions access from memory, there are no alignment restrictions.

Software may see performance penalties when unaligned accesses cross cacheline boundaries or vector-length naturally-aligned boundaries, so reasonable attempts to align commonly used data sets should continue to be pursued.

Atomic memory operation in Intel 64 and IA-32 architecture is guaranteed only for a subset of memory operand sizes and alignment scenarios. The guaranteed atomic operations are described in Section 7.1.1 of IA-32 Intel® Architecture Software Developer's Manual, Volumes 3A. AVX and FMA instructions do not introduce any new guaranteed atomic memory operations.

AVX-512 instructions may generate an #AC(0) fault on misaligned 4 or 8-byte memory references in Ring-3 when CR0.AM=1. 16, 32 and 64-byte memory references will not generate #AC(0) fault. See Table 2-4 for details.

Certain AVX-512 Foundation instructions always require 64-byte alignment (see the complete list of VEX and EVEX encoded instructions in Table 2-3). These instructions will #GP(0) if not aligned to 64-byte boundaries.

Table 2-3. SIMD Instructions Requiring Explicitly Aligned Memory

Require 16-byte alignment	Require 32-byte alignment	Require 64-byte alignment*
(V)MOVDQA xmm, m128	VMOVDQA ymm, m256	VMOVDQA zmm, m512
(V)MOVDQA m128, xmm	VMOVDQA m256, ymm	VMOVDQA m512, zmm
(V)MOVAPS xmm, m128	VMOVAPS ymm, m256	VMOVAPS zmm, m512
(V)MOVAPS m128, xmm	VMOVAPS m256, ymm	VMOVAPS m512, zmm
(V)MOVAPD xmm, m128	VMOVAPD ymm, m256	VMOVAPD zmm, m512
(V)MOVAPD m128, xmm	VMOVAPD m256, ymm	VMOVAPD m512, zmm
(V)MOVNTDQA xmm, m128	VMOVNTPS m256, ymm	VMOVNTPS m512, zmm
(V)MOVNTPS m128, xmm	VMOVNTPD m256, ymm	VMOVNTPD m512, zmm
(V)MOVNTPD m128, xmm	VMOVNTDQ m256, ymm	VMOVNTDQ m512, zmm
(V)MOVNTDQ m128, xmm	VMOVNTDQA ymm, m256	VMOVNTDQA zmm, m512

Table 2-4. Instructions Not Requiring Explicit Memory Alignment

(V)MOVDQU xmm, m128	VMOVDQU ymm, m256	VMOVDQU zmm, m512
(V)MOVDQU m128, m128	VMOVDQU m256, ymm	VMOVDQU m512, zmm
(V)MOVUPS xmm, m128	VMOVUPS ymm, m256	VMOVUPS zmm, m512
(V)MOVUPS m128, xmm	VMOVUPS m256, ymm	VMOVUPS m512, zmm
(V)MOVUPD xmm, m128	VMOVUPD ymm, m256	VMOVUPD zmm, m512
(V)MOVUPD m128, xmm	VMOVUPD m256, ymm	VMOVUPD m512, zmm

2.5 SIMD FLOATING-POINT EXCEPTIONS

AVX-512 instructions can generate SIMD floating-point exceptions (#XM) if embedded “suppress all exceptions” (SAE) in EVEX is not set. When SAE is not set, these instructions will respond to exception masks of MXCSR in the same way as VEX-encoded AVX instructions. When CR4.OSXMMEXCPT=0 any unmasked FP exceptions generate an Undefined Opcode exception (#UD).

2.6 INSTRUCTION EXCEPTION SPECIFICATION

Exception behavior of VEX-encoded AVX/AVX2 instructions are described in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*. Exception behavior of AVX-512 Foundation instructions and additional 512-bit extensions are described in Section 4.10, “Exception Classifications of EVEX-Encoded instructions” and Section 4.11, “Exception Classifications of Opmask instructions”.

2.7 CPUID INSTRUCTION

CPUID—CPU Identification

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F A2	CPUID	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

Description

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. This instruction operates the same in non-64-bit modes and 64-bit mode.

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers.[1] The instruction's output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

```
MOV EAX, 00H
CPUID
```

Table 2-5 shows information returned, depending on the initial value loaded into the EAX register. Table 2-6 shows the maximum CPUID input value recognized for each family of IA-32 processors on which CPUID is implemented.

Two types of information are returned: basic and extended function information. If a value is entered for CPUID.EAX is invalid for a particular processor, the data for the highest basic information leaf is returned. For example, using the Intel Core 2 Duo E6850 processor, the following is true:

```
CPUID.EAX = 05H (* Returns MONITOR/MWAIT leaf. *)
CPUID.EAX = 0AH (* Returns Architectural Performance Monitoring leaf. *)
CPUID.EAX = 0BH (* INVALID: Returns the same information as CPUID.EAX = 0AH. *)
CPUID.EAX = 80000008H (* Returns virtual/physical address size data. *)
CPUID.EAX = 8000000AH (* INVALID: Returns same information as CPUID.EAX = 0AH. *)
```

When CPUID returns the highest basic leaf information as a result of an invalid input EAX value, any dependence on input ECX value in the basic leaf is honored.

CPUID can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

See also:

"Serializing Instructions" in Chapter 8, "Multiple-Processor Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*

"Caching Translation Information" in Chapter 4, "Paging," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

1. On Intel 64 processors, CPUID clears the high 32 bits of the RAX/RBX/RCX/RDX registers in all modes.

Table 2-5. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0H	EAX EBX ECX EDX	Maximum Input Value for Basic CPUID Information (see Table 2-6) "Genu" "ntel" "inel"
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 2-2)
	EBX	Bits 7-0: Brand Index Bits 15-8: CLFLUSH line size (Value ∗ 8 = cache line size in bytes) Bits 23-16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31-24: Initial APIC ID
	ECX	Feature Information (see Figure 2-3 and Table 2-8)
	EDX	Feature Information (see Figure 2-4 and Table 2-9)
		NOTES: * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the maximum number of unique initial APIC IDs reserved for addressing different logical processors in a physical package.
02H	EAX EBX ECX EDX	Cache and TLB Information (see Table 2-10) Cache and TLB Information Cache and TLB Information Cache and TLB Information
03H	EAX	Reserved.
	EBX	Reserved.
	ECX	Bits 00-31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
	EDX	Bits 32-63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
		NOTES: Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature. See AP-485, *Intel Processor Identification and the CPUID Instruction* (Order Number 241618) for more information on PSN.
	CPUID leaves > 3 < 80000000 are visible only when IA32_MISC_ENABLES.BOOT_NT4[bit 22] = 0 (default).	
	Deterministic Cache Parameters Leaf	
04H		**NOTES:** Leaf 04H output depends on the initial value in ECX. See also: "INPUT EAX = 4: Returns Deterministic Cache Parameters for each level on page 2-26.
	EAX	Bits 4-0: Cache Type Field 0 = Null - No more caches 1 = Data Cache 2 = Instruction Cache 3 = Unified Cache 4-31 = Reserved

Table 2-5. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
		Bits 7-5: Cache Level (starts at 1) Bits 8: Self Initializing cache level (does not need SW initialization) Bits 9: Fully Associative cache Bits 13-10: Reserved Bits 25-14: Maximum number of addressable IDs for logical processors sharing this cache*, ** Bits 31-26: Maximum number of addressable IDs for processor cores in the physical package*, ***, ****
	EBX	Bits 11-00: L = System Coherency Line Size* Bits 21-12: P = Physical Line partitions* Bits 31-22: W = Ways of associativity*
	ECX	Bits 31-00: S = Number of Sets*
	EDX	Bit 0: WBINVD/INVD behavior on lower level caches Bit 10: Write-Back Invalidate/Invalidate 0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache 1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache. Bit 1: Cache Inclusiveness 0 = Cache is not inclusive of lower cache levels. 1 = Cache is inclusive of lower cache levels. Bit 2: Complex cache indexing 0 = Direct mapped cache 1 = A complex function is used to index the cache, potentially using all address bits. Bits 31-03: Reserved = 0 **NOTES:** * Add one to the return value to get the result. ** The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache *** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core_IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID. ****The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.
	MONITOR/MWAIT Leaf	
05H	EAX	Bits 15-00: Smallest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0
	EBX	Bits 15-00: Largest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0
	ECX	Bits 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported Bits 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled Bits 31 - 02: Reserved

Table 2-5. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 03 - 00: Number of C0* sub C-states supported using MWait Bits 07 - 04: Number of C1* sub C-states supported using MWAIT Bits 11 - 08: Number of C2* sub C-states supported using MWAIT Bits 15 - 12: Number of C3* sub C-states supported using MWAIT Bits 19 - 16: Number of C4* sub C-states supported using MWAIT Bits 31 - 20: Reserved = 0 **NOTE:** * The definition of C0 through C4 states for MWAIT extension are processor-specific C-states, not ACPI C-states.
	Thermal and Power Management Leaf	
06H	EAX	Bits 00: Digital temperature sensor is supported if set Bits 01: Intel Turbo Boost Technology is available Bits 31 - 02: Reserved
	EBX	Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor Bits 31 - 04: Reserved
	ECX	Bits 00: Hardware Coordination Feedback Capability (Presence of MCNT and ACNT MSRs). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of expected processor performance at frequency specified in CPUID Brand String Bits 02 - 01: Reserved = 0 Bit 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H) Bits 31 - 04: Reserved = 0
	EDX	Reserved = 0
	Structured Extended feature Leaf	
07H		**NOTES:** Leaf 07H main leaf (ECX = 0). IF leaf 07H is not supported, EAX=EBX=ECX=EDX=0
	EAX	Bits 31-0: Reports the maximum number sub-leaves that are supported in leaf 07H.

Table 2-5. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bit 00: FSGSBASE. Supports RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE if 1. Bits 02-01: Reserved Bit 03: BMI1 Bit 04: HLE Bit 05: AVX2 Bit 07: SMEP. Supports Supervisor Mode Execution Protection if 1. Bit 06: Reserved Bit 08: BMI2 Bit 09: ERMS Bit 10: INVPCID Bit 11: RTM Bits 13-12: Reserved Bit 14: Intel Memory Protection Extensions Bit 15: Reserved Bit 16: AVX512F Bit 17: Reserved Bit 18: RDSEED Bit 19: ADX Bit 20: SMAP Bits 24-21: Reserved Bit 25: Intel Processor Trace Bit 26: AVX512PF Bit 27: AVX512ER Bit 28: AVX512CD Bit 29: SHA Bits 31-30: Reserved
	ECX	Bit 31-0: Reserved
	EDX	Bit 31-0: Reserved.
	Structured Extended Feature Enumeration Sub-leaves (EAX = 07H, ECX = n, n > 1)	
07H		**NOTES:** Leaf 07H output depends on the initial value in ECX. If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0.
	EAX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	EBX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	ECX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	EDX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	Direct Cache Access Information Leaf	
09H	EAX EBX ECX EDX	Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H) Reserved Reserved Reserved
	Architectural Performance Monitoring Leaf	
0AH	EAX	Bits 07 - 00: Version ID of architectural performance monitoring Bits 15- 08: Number of general-purpose performance monitoring counter per logical processor Bits 23 - 16: Bit width of general-purpose, performance monitoring counter Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events

Table 2-5. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bit 00: Core cycle event not available if 1 Bit 01: Instruction retired event not available if 1 Bit 02: Reference cycles event not available if 1 Bit 03: Last-level cache reference event not available if 1 Bit 04: Last-level cache misses event not available if 1 Bit 05: Branch instruction retired event not available if 1 Bit 06: Branch mispredict retired event not available if 1 Bits 31- 07: Reserved = 0
	ECX	Reserved = 0 Bits 04 - 00: Number of fixed-function performance counters (if Version ID > 1) Bits 12- 05: Bit width of fixed-function performance counters (if Version ID > 1)
	EDX	Reserved = 0
	Extended Topology Enumeration Leaf	
0BH		**NOTES:** Most of Leaf 0BH output depends on the initial value in ECX. The EDX output of leaf 0BH is always valid and does not vary with input value in ECX Output value in ECX[7:0] always equals input value in ECX[7:0]. For sub-leaves that returns an invalid level-type of 0 in ECX[15:8]; EAX and EBX will return 0 If an input value N in ECX returns the invalid level-type of 0 in ECX[15:8], other input values with ECX > N also return 0 in ECX[15:8]
	EAX	Bits 04-00: Number of bits to shift right on x2APIC ID to get a unique topology ID of the next level type*. All logical processors with the same next level ID share current level. Bits 31-5: Reserved.
	EBX	Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**. Bits 31- 16: Reserved.
	ECX	Bits 07 - 00: Level number. Same value in ECX input Bits 15 - 08: Level type***. Bits 31 - 16: Reserved.
	EDX	Bits 31- 0: x2APIC ID the current logical processor.
		NOTES: * Software should use this field (EAX[4:0]) to enumerate processor topology of the system. ** Software must not use EBX[15:0] to enumerate processor topology of the system. This value in this field (EBX[15:0]) is only intended for display/diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations. *** The value of the "level type" field is not related to level numbers in any way, higher "level type" values do not mean higher levels. Level type field has the following encoding: 0: invalid 1: SMT 2: Core 3-255: Reserved
	Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)	
0DH		**NOTES:** Leaf 0DH main leaf (ECX = 0).

Table 2-5. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EAX	Bits 31-00: Reports the valid bit fields of the lower 32 bits of the XFEATURE_ENABLED_MASK register. If a bit is 0, the corresponding bit field in XCR0 is reserved. Bit 00: legacy x87 Bit 01: 128-bit SSE Bit 02: 256-bit AVX
	EBX	Bits 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCR0. May be different than ECX if some features at the end of the XSAVE save area are not enabled.
	ECX	Bit 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e all the valid bit fields in XCR0.
	EDX	Bit 31-0: Reports the valid bit fields of the upper 32 bits of the XCR0 register. If a bit is 0, the corresponding bit field in XCR0 is reserved
	Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)	
	EAX	Bit 00: XSAVEOPT is available; Bits 31-1: Reserved
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
	Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n > 1)	
0DH		**NOTES:** Leaf 0DH output depends on the initial value in ECX. If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Each valid sub-leaf index maps to a valid bit in the XCR0 register starting at bit position 2
	EAX	Bits 31-0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, *n*. This field reports 0 if the sub-leaf index, *n*, is invalid*.
	EBX	Bits 31-0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area. This field reports 0 if the sub-leaf index, *n*, is invalid*.
	ECX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	EDX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
		*The highest valid sub-leaf index, *n*, is (POPCNT(CPUID.(EAX=0D, ECX=0):EAX) + POPCNT(CPUID.(EAX=0D, ECX=0):EDX) - 1)
	Intel Processor Trace Enumeration Main Leaf (EAX = 14H, ECX = 0)	
14H		**NOTES:** Leaf 14H main leaf (ECX = 0).
	EAX	Bits 31-0: Reports the maximum number sub-leaves that are supported in leaf 14H.
	EBX	Bit 00: If 1, Indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed. Bits 31- 01: Reserved

Table 2-5. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	ECX	Bit 00: If 1, Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme; IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed. Bit 01: If 1, ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. Bit 30:02: Reserved Bit 31: If 1, Generated packets which contain IP payloads have LIP values, which include the CS base component.
	EDX	Bits 31- 00: Reserved
	Extended Function CPUID Information	
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information (see Table 2-6).
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
80000001H	EAX	Extended Processor Signature and Feature Bits.
	EBX	Reserved
	ECX	Bit 0: LAHF/SAHF available in 64-bit mode Bits 4-1: Reserved Bit 5: LZCNT available Bits 7-6 Reserved Bit 8: PREFETCHW Bits 31-9: Reserved
	EDX	Bits 10-0: Reserved Bit 11: SYSCALL/SYSRET available (when in 64-bit mode) Bits 19-12: Reserved = 0 Bit 20: Execute Disable Bit available Bits 28-21: Reserved = 0 Bit 29: Intel® 64 Architecture available if 1 Bits 31-30: Reserved = 0
80000002H	EAX	Processor Brand String
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000003H	EAX	Processor Brand String Continued
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000004H	EAX	Processor Brand String Continued
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000005H	EAX	Reserved = 0
	EBX	Reserved = 0
	ECX	Reserved = 0
	EDX	Reserved = 0
80000006H	EAX	Reserved = 0
	EBX	Reserved = 0

Table 2-5. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	ECX	Bits 7-0: Cache Line size in bytes Bits 15-12: L2 Associativity field * Bits 31-16: Cache size in 1K units
	EDX	Reserved = 0
		NOTES: * L2 associativity field encodings: 00H - Disabled 01H - Direct mapped 02H - 2-way 04H - 4-way 06H - 8-way 08H - 16-way 0FH - Fully associative
80000007H	EAX	Reserved = 0
	EBX	Reserved = 0
	ECX	Reserved = 0
	EDX	Reserved = 0
80000008H	EAX	Virtual/Physical Address size Bits 7-0: #Physical Address Bits* Bits 15-8: #Virtual Address Bits Bits 31-16: Reserved = 0
	EBX	Reserved = 0
	ECX	Reserved = 0
	EDX	Reserved = 0
		NOTES: * If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field.

INPUT EAX = 0: Returns CPUID's Highest Value for Basic Processor Information and the Vendor Identification String

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register (see Table 2-6) and is processor specific.

A vendor identification string is also returned in EBX, EDX, and ECX. For Intel processors, the string is "GenuineIntel" and is expressed:

EBX ← 756e6547h (* "Genu", with G in the low 4 bits of BL *)
EDX ← 49656e69h (* "inel", with i in the low 4 bits of DL *)
ECX ← 6c65746eh (* "ntel", with n in the low 4 bits of CL *)

INPUT EAX = 80000000H: Returns CPUID's Highest Value for Extended Processor Information

When CPUID executes with EAX set to 0, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register (see Table 2-6) and is processor specific.

Table 2-6. Highest CPUID Source Operand for Intel 64 and IA-32 Processors

Intel 64 or IA-32 Processors	Highest Value in EAX	
	Basic Information	Extended Function Information
Earlier Intel486 Processors	CPUID Not Implemented	CPUID Not Implemented
Later Intel486 Processors and Pentium Processors	01H	Not Implemented
Pentium Pro and Pentium II Processors, Intel® Celeron® Processors	02H	Not Implemented
Pentium III Processors	03H	Not Implemented
Pentium 4 Processors	02H	80000004H
Intel Xeon Processors	02H	80000004H
Pentium M Processor	02H	80000004H
Pentium 4 Processor supporting Hyper-Threading Technology	05H	80000008H
Pentium D Processor (8xx)	05H	80000008H
Pentium D Processor (9xx)	06H	80000008H
Intel Core Duo Processor	0AH	80000008H
Intel Core 2 Duo Processor	0AH	80000008H
Intel Xeon Processor 3000, 5100, 5300 Series	0AH	80000008H
Intel Xeon Processor 3000, 5100, 5200, 5300, 5400 Series	0AH	80000008H
Intel Core 2 Duo Processor 8000 Series	0DH	80000008H
Intel Xeon Processor 5200, 5400 Series	0AH	80000008H

IA32_BIOS_SIGN_ID Returns Microcode Update Signature

For processors that support the microcode update facility, the IA32_BIOS_SIGN_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

INPUT EAX = 1: Returns Model, Family, Stepping Information

When CPUID executes with EAX set to 1, version information is returned in EAX (see Figure 2-2). For example: model, family, and processor type for the Intel Xeon processor 5100 series is as follows:

- Model — 1111B
- Family — 0101B
- Processor Type — 00B

See Table 2-7 for available processor type values. Stepping IDs are provided as needed.

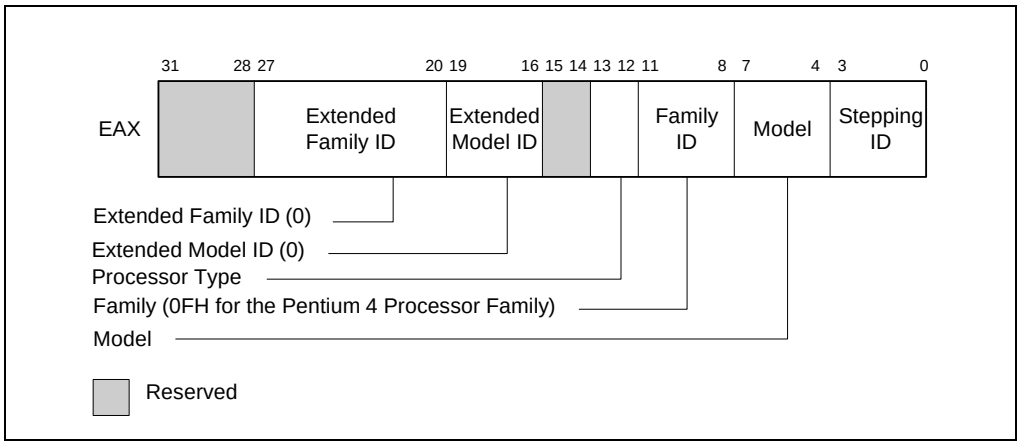


Figure 2-2. Version Information Returned by CPUID in EAX

Table 2-7. Processor Type Field

Type	Encoding
Original OEM Processor	00B
Intel OverDrive® Processor	01B
Dual processor (not applicable to Intel486 processors)	10B
Intel reserved	11B

NOTE

See "Caching Translation Information" in Chapter 4, "Paging," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, and Chapter 16 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for information on identifying earlier IA-32 processors.

The Extended Family ID needs to be examined only when the Family ID is 0FH. Integrate the fields into a display using the following rule:

```
IF Family_ID ≠ 0FH
    THEN Displayed_Family = Family_ID;
    ELSE Displayed_Family = Extended_Family_ID + Family_ID;
    (* Right justify and zero-extend 4-bit field. *)
FI;
(* Show Display_Family as HEX field. *)
```

The Extended Model ID needs to be examined only when the Family ID is 06H or 0FH. Integrate the field into a display using the following rule:

```
IF (Family_ID = 06H or Family_ID = 0FH)
    THEN Displayed_Model = (Extended_Model_ID << 4) + Model_ID;
    (* Right justify and zero-extend 4-bit field; display Model_ID as HEX field.*)
    ELSE Displayed_Model = Model_ID;
FI;
(* Show Display_Model as HEX field. *)
```

INPUT EAX = 1: Returns Additional Information in EBX

When CPUID executes with EAX set to 1, additional information is returned to the EBX register:

- Brand index (low byte of EBX) — this number provides an entry into a brand string table that contains brand strings for IA-32 processors. More information about this field is provided later in this section.
- CLFLUSH instruction cache line size (second byte of EBX) — this number indicates the size of the cache line flushed with CLFLUSH instruction in 8-byte increments. This field was introduced in the Pentium 4 processor.
- Local APIC ID (high byte of EBX) — this number is the 8-bit ID that is assigned to the local APIC on the processor during power up. This field was introduced in the Pentium 4 processor.

INPUT EAX = 1: Returns Feature Information in ECX and EDX

When CPUID executes with EAX set to 1, feature information is returned in ECX and EDX.

- Figure 2-3 and Table 2-8 show encodings for ECX.
- Figure 2-4 and Table 2-9 show encodings for EDX.

For all feature flags, a 1 indicates that the feature is supported. Use Intel to properly interpret feature flags.

NOTE

Software must confirm that a processor feature is present using feature flags returned by CPUID prior to using the feature. Software should not depend on future offerings retaining all features.

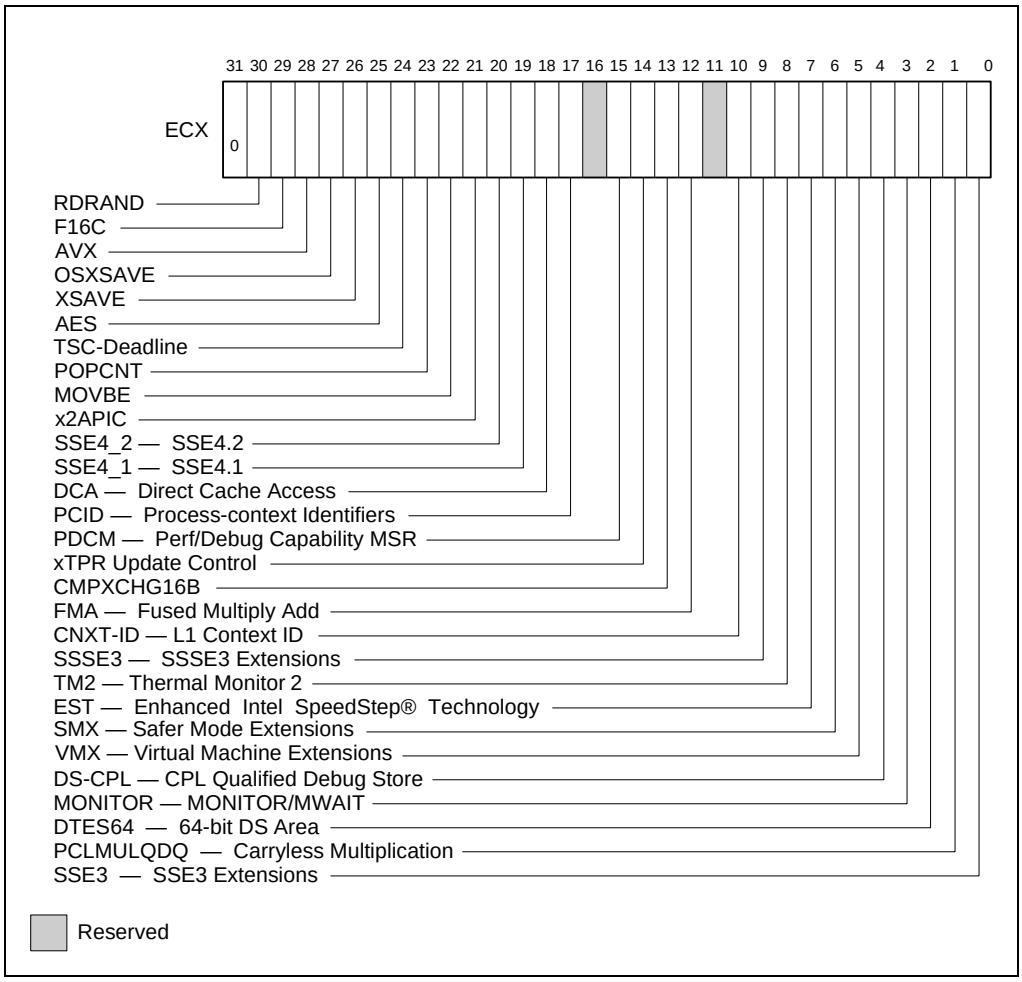


Figure 2-3. Feature Information Returned in the ECX Register

Table 2-8. Feature Information Returned in the ECX Register

Bit #	Mnemonic	Description
0	SSE3	**Streaming SIMD Extensions 3 (SSE3)**. A value of 1 indicates the processor supports this technology.
1	PCLMULQDQ	A value of 1 indicates the processor supports PCLMULQDQ instruction.
2	DTES64	**64-bit DS Area**. A value of 1 indicates the processor supports DS area using 64-bit layout.
3	MONITOR	**MONITOR/MWAIT**. A value of 1 indicates the processor supports this feature.
4	DS-CPL	**CPL Qualified Debug Store**. A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	Virtual Machine Extensions. A value of 1 indicates that the processor supports this technology.
6	SMX	Safer Mode Extensions. A value of 1 indicates that the processor supports this technology. See Chapter 5, "Safer Mode Extensions Reference".
7	EST	**Enhanced Intel SpeedStep® technology**. A value of 1 indicates that the processor supports this technology.
8	TM2	**Thermal Monitor 2**. A value of 1 indicates whether the processor supports this technology.
9	SSSE3	A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor.
10	CNXT-ID	**L1 Context ID**. A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
11	Reserved	Reserved
12	FMA	A value of 1 indicates the processor supports FMA extensions using YMM state.
13	CMPXCHG16B	**CMPXCHG16B Available**. A value of 1 indicates that the feature is available.
14	xTPR Update Control	**xTPR Update Control**. A value of 1 indicates that the processor supports changing IA32_MISC_ENABLES[bit 23].
15	PDCM	**Perfmon and Debug Capability**: A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	**Process-context identifiers.** A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4.2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor's local APIC timer supports one-shot operation using a TSC deadline value.
25	AES	A value of 1 indicates that the processor supports the AES instruction.
26	XSAVE	A value of 1 indicates that the processor supports the XFEATURE_ENABLED_MASK register and XSAVE/XRSTOR/XSETBV/XGETBV instructions to manage processor extended states.
27	OSXSAVE	A value of 1 indicates that the OS has enabled support for using XGETBV/XSETBV instructions to query processor extended states.
28	AVX	A value of 1 indicates that processor supports AVX instructions operating on 256-bit YMM state, and three-operand encoding of 256-bit and 128-bit SIMD instructions.
29	F16C	A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.
30	RDRAND	A value of 1 indicates that processor supports RDRAND instruction.
31	Not Used	Always return 0

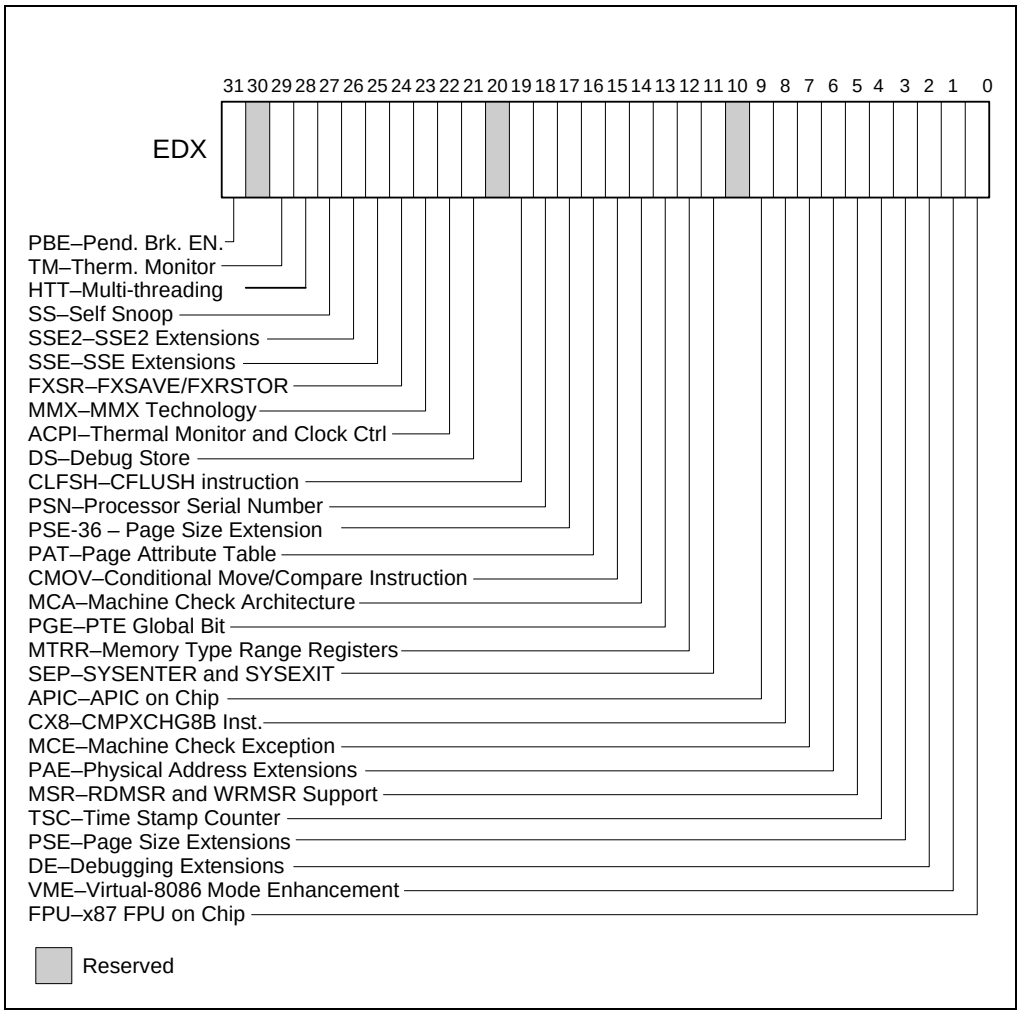


Figure 2-4. Feature Information Returned in the EDX Register

Table 2-9. More on Feature Information Returned in the EDX Register

Bit #	Mnemonic	Description
0	FPU	**floating-point Unit On-Chip.** The processor contains an x87 FPU.
1	VME	**Virtual 8086 Mode Enhancements.** Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
2	DE	**Debugging Extensions.** Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.
3	PSE	**Page Size Extension.** Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
4	TSC	**Time Stamp Counter.** The RDTSC instruction is supported, including CR4.TSD for controlling privilege.
5	MSR	**Model Specific Registers RDMSR and WRMSR Instructions.** The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.

Table 2-9. More on Feature Information Returned in the EDX Register (Contd.)

Bit #	Mnemonic	Description
6	PAE	**Physical Address Extension.** Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1. The actual number of address bits beyond 32 is not defined, and is implementation specific.
7	MCE	**Machine Check Exception**. Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.
8	CX8	**CMPXCHG8B Instruction.** The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
9	APIC	**APIC On-Chip.** The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).
10	Reserved	Reserved
11	SEP	**SYSENTER and SYSEXIT Instructions.** The SYSENTER and SYSEXIT and associated MSRs are supported.
12	MTRR	**Memory Type Range Registers.** MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.
13	PGE	**PTE Global Bit.** The global bit in page directory entries (PDEs) and page table entries (PTEs) is supported, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.
14	MCA	**Machine Check Architecture.** The Machine Check Architecture, which provides a compatible mechanism for error reporting in P6 family, Pentium 4, Intel Xeon processors, and future processors, is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.
15	CMOV	**Conditional Move Instructions.** The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported
16	PAT	**Page Attribute Table.** Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory on a 4K granularity through a linear address.
17	PSE-36	**36-Bit Page Size Extension.** Extended 4-MByte pages that are capable of addressing physical memory beyond 4 GBytes are supported. This feature indicates that the upper four bits of the physical address of the 4-MByte page is encoded by bits 13-16 of the page directory entry.
18	PSN	**Processor Serial Number.** The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	**CLFLUSH Instruction.** CLFLUSH Instruction is supported.
20	Reserved	Reserved
21	DS	**Debug Store.** The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and precise event-based sampling (PEBS) facilities (see Chapter 17, "Debugging, Branch Profiles and Time-Stamp Counter," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).
22	ACPI	**Thermal Monitor and Software Controlled Clock Facilities.** The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	**Intel MMX Technology.** The processor supports the Intel MMX technology.
24	FXSR	**FXSAVE and FXRSTOR Instructions.** The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating-point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.

Table 2-9. More on Feature Information Returned in the EDX Register (Contd.)

Bit #	Mnemonic	Description
25	SSE	**SSE.** The processor supports the SSE extensions.
26	SSE2	**SSE2.** The processor supports the SSE2 extensions.
27	SS	**Self Snoop.** The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.
28	HTT	**Multi-Threading.** The physical processor package is capable of supporting more than one logical processor.
29	TM	**Thermal Monitor.** The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30	Reserved	Reserved
31	PBE	**Pending Break Enable.** The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability.

INPUT EAX = 2: Cache and TLB Information Returned in EAX, EBX, ECX, EDX

When CPUID executes with EAX set to 2, the processor returns information about the processor's internal caches and TLBs in the EAX, EBX, ECX, and EDX registers.

The encoding is as follows:

- The least-significant byte in register EAX (register AL) indicates the number of times the CPUID instruction must be executed with an input value of 2 to get a complete description of the processor's caches and TLBs. The first member of the family of Pentium 4 processors will return a 1.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. Table 2-10 shows the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache or TLB types. The descriptors may appear in any order.

Table 2-10. Encoding of Cache and TLB Descriptors

Descriptor Value	Cache or TLB Description
00H	Null descriptor
01H	Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries
02H	Instruction TLB: 4 MByte pages, 4-way set associative, 2 entries
03H	Data TLB: 4 KByte pages, 4-way set associative, 64 entries
04H	Data TLB: 4 MByte pages, 4-way set associative, 8 entries
05H	Data TLB1: 4 MByte pages, 4-way set associative, 32 entries
06H	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size
08H	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size
0AH	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size
0BH	Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries
0CH	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size
22H	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
25H	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
29H	3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
2CH	1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size

Table 2-10. Encoding of Cache and TLB Descriptors (Contd.)

Descriptor Value	Cache or TLB Description
30H	1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size
40H	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size
42H	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size
43H	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size
44H	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size
45H	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size
46H	3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size
47H	3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size
49H	3rd-level cache: 4MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H); 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size
4AH	3rd-level cache: 6MByte, 12-way set associative, 64 byte line size
4BH	3rd-level cache: 8MByte, 16-way set associative, 64 byte line size
4CH	3rd-level cache: 12MByte, 12-way set associative, 64 byte line size
4DH	3rd-level cache: 16MByte, 16-way set associative, 64 byte line size
4EH	2nd-level cache: 6MByte, 24-way set associative, 64 byte line size
50H	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries
51H	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries
52H	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries
56H	Data TLB0: 4 MByte pages, 4-way set associative, 16 entries
57H	Data TLB0: 4 KByte pages, 4-way associative, 16 entries
5BH	Data TLB: 4 KByte and 4 MByte pages, 64 entries
5CH	Data TLB: 4 KByte and 4 MByte pages,128 entries
5DH	Data TLB: 4 KByte and 4 MByte pages,256 entries
60H	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size
66H	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size
67H	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size
68H	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size
70H	Trace cache: 12 K-µop, 8-way set associative
71H	Trace cache: 16 K-µop, 8-way set associative
72H	Trace cache: 32 K-µop, 8-way set associative
78H	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size
79H	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector
7DH	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size
7FH	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size
82H	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size
83H	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size
84H	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size

Table 2-10. Encoding of Cache and TLB Descriptors (Contd.)

Descriptor Value	Cache or TLB Description
85H	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size
86H	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size
87H	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size
B0H	Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries
B1H	Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries
B3H	Data TLB: 4 KByte pages, 4-way set associative, 128 entries
B4H	Data TLB1: 4 KByte pages, 4-way associative, 256 entries
F0H	64-Byte prefetching
F1H	128-Byte prefetching

Example 2-1. Example of Cache and TLB Interpretation

The first member of the family of Pentium 4 processors returns the following information about caches and TLBs when the CPUID executes with an input value of 2:

```
EAX     66 5B 50 01H
EBX     0H
ECX     0H
EDX     00 7A 70 00H
```

Which means:

- The least-significant byte (byte 0) of register EAX is set to 01H. This indicates that CPUID needs to be executed once with an input value of 2 to retrieve complete information about caches and TLBs.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor has:
  - 50H - a 64-entry instruction TLB, for mapping 4-KByte and 2-MByte or 4-MByte pages.
  - 5BH - a 64-entry data TLB, for mapping 4-KByte and 4-MByte pages.
  - 66H - an 8-KByte 1st level data cache, 4-way set associative, with a 64-Byte cache line size.
- The descriptors in registers EBX and ECX are valid, but contain NULL descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor has:
  - 00H - NULL descriptor.
  - 70H - Trace cache: 12 K-μop, 8-way set associative.
  - 7AH - a 256-KByte 2nd level cache, 8-way set associative, with a sectored, 64-byte cache line size.
  - 00H - NULL descriptor.

INPUT EAX = 4: Returns Deterministic Cache Parameters for Each Level

When CPUID executes with EAX set to 4 and ECX contains an index value, the processor returns encoded data that describe a set of deterministic cache parameters (for the cache level associated with the input in ECX). Valid index values start from 0.

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 2-5.

The CPUID leaf 4 also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=4 and ECX=0 and use it as part of the topology enumeration algorithm described in Chapter 8,

"Multiple-Processor Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

INPUT EAX = 5: Returns MONITOR and MWAIT Features

When CPUID executes with EAX set to 5, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 2-5.

INPUT EAX = 6: Returns Thermal and Power Management Features

When CPUID executes with EAX set to 6, the processor returns information about thermal and power management features. See Table 2-5.

INPUT EAX = 7: Returns Structured Extended Feature Enumeration Information

When CPUID executes with EAX set to 7 and ECX = 0, the processor returns information about the maximum number of sub-leaves that contain extended feature flags. See Table 2-5.

When CPUID executes with EAX set to 7 and ECX = n (n > 1and less than the number of non-zero bits in CPUID.(EAX=07H, ECX= 0H).EAX, the processor returns information about extended feature flags. See Table 2-5. In sub-leaf 0, only EAX has the number of sub-leaves. In sub-leaf 0, EBX, ECX & EDX all contain extended feature flags.

Table 2-11. Structured Extended Feature Leaf, Function 0, EBX Register

Bit #	Mnemonic	Description
0	RWFSGSBASE	A value of 1 indicates the processor supports RD/WR FSGSBASE instructions
1-31	Reserved	Reserved

INPUT EAX = 9: Returns Direct Cache Access Information

When CPUID executes with EAX set to 9, the processor returns information about Direct Cache Access capabilities. See Table 2-5.

INPUT EAX = 10: Returns Architectural Performance Monitoring Features

When CPUID executes with EAX set to 10, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 2-5) is greater than Pn 0. See Table 2-5.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 17, "Debugging, Branch Profiles and Time-Stamp Counter," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

INPUT EAX = 11: Returns Extended Topology Information

When CPUID executes with EAX set to 11, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 0BH by verifying (a) the highest leaf index supported by CPUID is >= 0BH, and (b) CPUID.0BH:EBX[15:0] reports a non-zero value.

INPUT EAX = 13: Returns Processor Extended States Enumeration Information

When CPUID executes with EAX set to 13 and ECX = 0, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 2-5.

When CPUID executes with EAX set to 13 and ECX = n (n > 1and less than the number of non-zero bits in CPUID.(EAX=0DH, ECX= 0H).EAX and CPUID.(EAX=0DH, ECX= 0H).EDX), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 2-5.

INPUT EAX = 20: Returns Intel Processor Trace Enumeration Information

When CPUID executes with EAX set to 20and ECX = 0, the processor returns information about Intel Processor Trace extensions. See Table 2-5.

When CPUID executes with EAX set to 20 and ECX = n (n > 1and less than the number of non-zero bits in CPUID.(EAX=14H, ECX= 0H).EAX and CPUID.(EAX=0DH, ECX= 0H).EDX), the processor returns information about packet generation in Intel Processor Trace. See Table 2-5.

METHODS FOR RETURNING BRANDING INFORMATION

Use the following techniques to access branding information:

1. Processor brand string method; this method also returns the processor's maximum operating frequency
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: "Identification of Earlier IA-32 Processors" in Chapter 16 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The Processor Brand String Method

Figure 2-5 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

This method (introduced with Pentium 4 processors) returns an ASCII brand identification string and the maximum operating frequency of the processor to the EAX, EBX, ECX, and EDX registers.

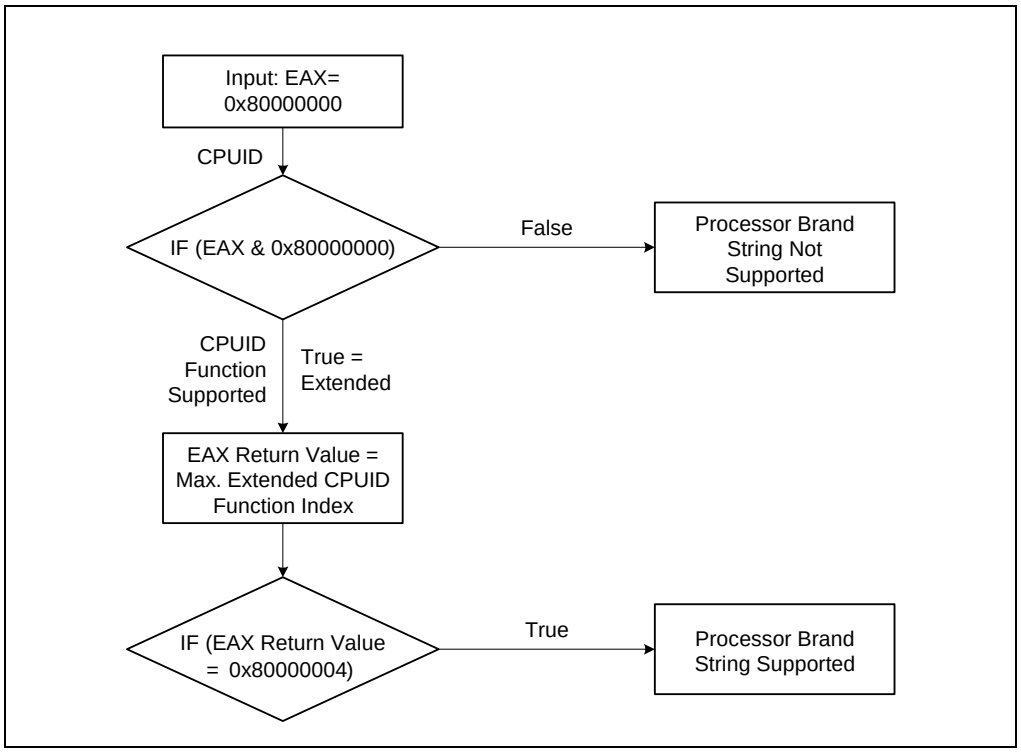


Figure 2-5. Determination of Support for the Processor Brand String

How Brand Strings Work

To use the brand string method, execute CPUID with EAX input of 8000002H through 80000004H. For each input value, CPUID returns 16 ASCII characters using EAX, EBX, ECX, and EDX. The returned string will be NULL-terminated.

Table 2-12 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 2-12. Processor Brand String Returned with Pentium 4 Processor

EAX Input Value	Return Values	ASCII Equivalent
80000002H	EAX = 20202020H	" "
	EBX = 20202020H	" "
	ECX = 20202020H	" "
	EDX = 6E492020H	"nI "
80000003H	EAX = 286C6574H	"(let"
	EBX = 50202952H	"P)R"
	ECX = 69746E65H	"itne"
	EDX = 52286D75H	"R(mu"
80000004H	EAX = 20342029H	" 4)"
	EBX = 20555043H	" UPC"
	ECX = 30303531H	"0051"
	EDX = 007A484DH	"\0zHM"

Extracting the Maximum Processor Frequency from Brand Strings

Figure 2-6 provides an algorithm which software can use to extract the maximum processor operating frequency from the processor brand string.

NOTE

When a frequency is given in a brand string, it is the maximum qualified frequency of the processor, not the frequency at which the processor is currently running.

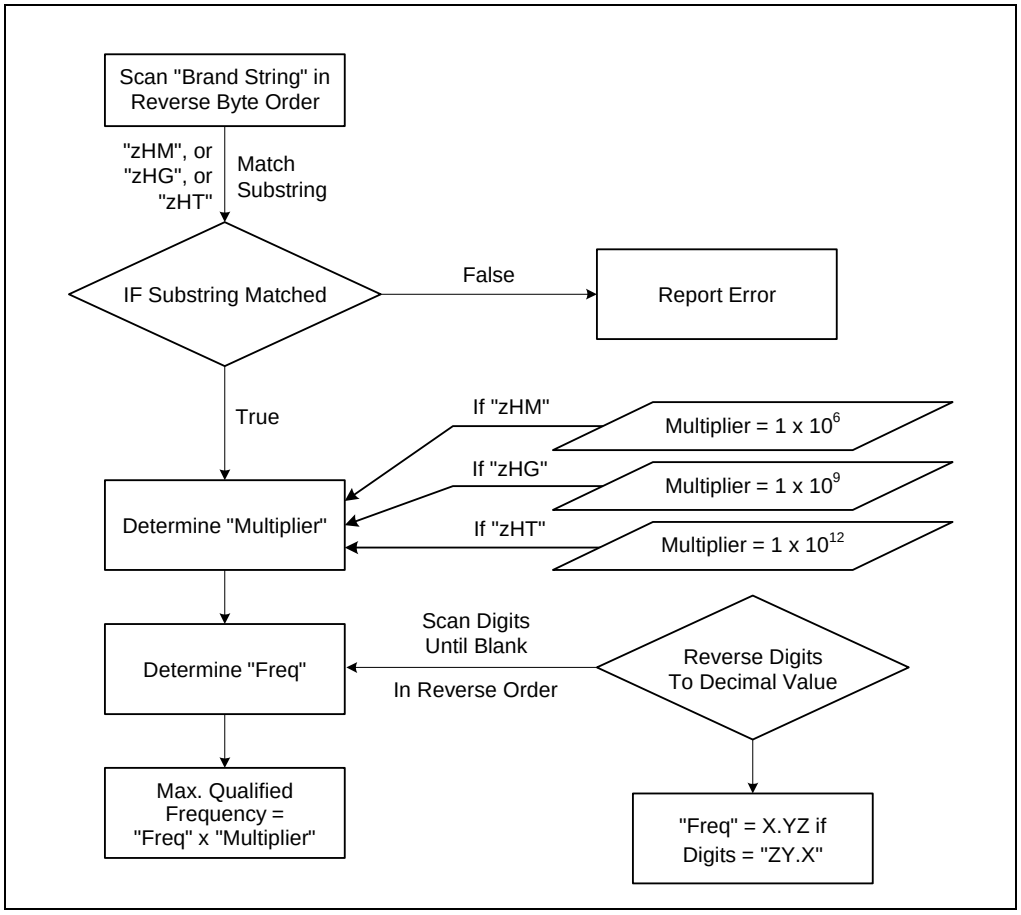


Figure 2-6. Algorithm for Extracting Maximum Processor Frequency

The Processor Brand Index Method

The brand index method (introduced with Pentium® III Xeon® processors) provides an entry point into a brand identification table that is maintained in memory by system software and is accessible from system- and user-level code. In this table, each brand index is associate with an ASCII brand identification string that identifies the official Intel family and model number of a processor.

When CPUID executes with EAX set to 1, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that do not support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 2-13 shows brand indices that have identification strings associated with them.

Table 2-13. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings

Brand Index	Brand String
00H	This processor does not support the brand identification feature
01H	Intel(R) Celeron(R) processor[1]

Table 2-13. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings

02H	Intel(R) Pentium(R) III processor[1]
03H	Intel(R) Pentium(R) III Xeon(R) processor; If processor signature = 000006B1h, then Intel(R) Celeron(R) processor
04H	Intel(R) Pentium(R) III processor
06H	Mobile Intel(R) Pentium(R) III processor-M
07H	Mobile Intel(R) Celeron(R) processor[1]
08H	Intel(R) Pentium(R) 4 processor
09H	Intel(R) Pentium(R) 4 processor
0AH	Intel(R) Celeron(R) processor[1]
0BH	Intel(R) Xeon(R) processor; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor MP
0CH	Intel(R) Xeon(R) processor MP
0EH	Mobile Intel(R) Pentium(R) 4 processor-M; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor
0FH	Mobile Intel(R) Celeron(R) processor[1]
11H	Mobile Genuine Intel(R) processor
12H	Intel(R) Celeron(R) M processor
13H	Mobile Intel(R) Celeron(R) processor[1]
14H	Intel(R) Celeron(R) processor
15H	Mobile Genuine Intel(R) processor
16H	Intel(R) Pentium(R) M processor
17H	Mobile Intel(R) Celeron(R) processor[1]
18H - 0FFH	RESERVED

NOTES:
1.Indicates versions of these processors that were introduced after the Pentium III

IA-32 Architecture Compatibility

CPUID is not supported in early models of the Intel486 processor or in any IA-32 processor earlier than the Intel486 processor.

Operation

```
IA32_BIOS_SIGN_ID MSR ← Update with installed microcode revision number;

CASE (EAX) OF
    EAX = 0:
        EAX ← Highest basic function input value understood by CPUID;
        EBX ← Vendor identification string;
        EDX ← Vendor identification string;
        ECX ← Vendor identification string;
    BREAK;
    EAX = 1H:
        EAX[3:0] ← Stepping ID;
        EAX[7:4] ← Model;
        EAX[11:8] ← Family;
        EAX[13:12] ← Processor type;
        EAX[15:14] ← Reserved;
        EAX[19:16] ← Extended Model;
```

```
        EAX[27:20] ← Extended Family;
        EAX[31:28] ← Reserved;
        EBX[7:0] ← Brand Index; (* Reserved if the value is zero. *)
        EBX[15:8] ← CLFLUSH Line Size;
        EBX[16:23] ← Reserved; (* Number of threads enabled = 2 if MT enable fuse set. *)
        EBX[24:31] ← Initial APIC ID;
        ECX ← Feature flags; (* See Figure 2-3. *)
        EDX ← Feature flags; (* See Figure 2-4. *)
    BREAK;
    EAX = 2H:
        EAX ← Cache and TLB information;
        EBX ← Cache and TLB information;
        ECX ← Cache and TLB information;
        EDX ← Cache and TLB information;
    BREAK;
    EAX = 3H:
        EAX ← Reserved;
        EBX ← Reserved;
        ECX ← ProcessorSerialNumber[31:0];
        (* Pentium lll processors only, otherwise reserved. *)
        EDX ← ProcessorSerialNumber[63:32];
        (* Pentium lll processors only, otherwise reserved. *
    BREAK
    EAX = 4H:
        EAX ← Deterministic Cache Parameters Leaf; (* See Table 2-5. *)
        EBX ← Deterministic Cache Parameters Leaf;
        ECX ← Deterministic Cache Parameters Leaf;
        EDX ← Deterministic Cache Parameters Leaf;
    BREAK;
    EAX = 5H:
        EAX ← MONITOR/MWAIT Leaf; (* See Table 2-5. *)
        EBX ← MONITOR/MWAIT Leaf;
        ECX ← MONITOR/MWAIT Leaf;
        EDX ← MONITOR/MWAIT Leaf;
    BREAK;
    EAX = 6H:
        EAX ← Thermal and Power Management Leaf; (* See Table 2-5. *)
        EBX ← Thermal and Power Management Leaf;
        ECX ← Thermal and Power Management Leaf;
        EDX ← Thermal and Power Management Leaf;
    BREAK;
    EAX = 7H:
        EAX ← Structured Extended Feature Leaf; (* See Table 2-5. *);
        EBX ← Structured Extended Feature Leaf;
        ECX ← Structured Extended Feature Leaf;
        EDX ← Structured Extended Feature Leaf;
    BREAK;
    EAX = 8H:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    EAX = 9H:
```

```
        EAX ← Direct Cache Access Information Leaf; (* See Table 2-5. *)
        EBX ← Direct Cache Access Information Leaf;
        ECX ← Direct Cache Access Information Leaf;
        EDX ← Direct Cache Access Information Leaf;
    BREAK;
    EAX = AH:
        EAX ← Architectural Performance Monitoring Leaf; (* See Table 2-5. *)
        EBX ← Architectural Performance Monitoring Leaf;
        ECX ← Architectural Performance Monitoring Leaf;
        EDX ← Architectural Performance Monitoring Leaf;
        BREAK
    EAX = BH:
        EAX ← Extended Topology Enumeration Leaf; (* See Table 2-5. *)
        EBX ← Extended Topology Enumeration Leaf;
        ECX ← Extended Topology Enumeration Leaf;
        EDX ← Extended Topology Enumeration Leaf;
    BREAK;
    EAX = CH:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    EAX = DH:
        EAX ← Processor Extended State Enumeration Leaf; (* See Table 2-5. *)
        EBX ← Processor Extended State Enumeration Leaf;
        ECX ← Processor Extended State Enumeration Leaf;
        EDX ← Processor Extended State Enumeration Leaf;
    BREAK;
    EAX = 14H:
        EAX ← Intel Processor Trace Enumeration Leaf; (* See Table 2-5. *)
        EBX ← Intel Processor Trace Enumeration Leaf;
        ECX ← Intel Processor Trace Enumeration Leaf;
        EDX ← Intel Processor Trace Enumeration Leaf;
    BREAK;
BREAK;
    EAX = 80000000H:
        EAX ← Highest extended function input value understood by CPUID;
        EBX ← Reserved;
        ECX ← Reserved;
        EDX ← Reserved;
    BREAK;
    EAX = 80000001H:
        EAX ← Reserved;
        EBX ← Reserved;
        ECX ← Extended Feature Bits (* See Table 2-5.*);
        EDX ← Extended Feature Bits (* See Table 2-5. *);
    BREAK;
    EAX = 80000002H:
        EAX ← Processor Brand String;
        EBX ← Processor Brand String, continued;
        ECX ← Processor Brand String, continued;
        EDX ← Processor Brand String, continued;
    BREAK;
```

```
    EAX = 80000003H:
        EAX ← Processor Brand String, continued;
        EBX ← Processor Brand String, continued;
        ECX ← Processor Brand String, continued;
        EDX ← Processor Brand String, continued;
    BREAK;
    EAX = 80000004H:
        EAX ← Processor Brand String, continued;
        EBX ← Processor Brand String, continued;
        ECX ← Processor Brand String, continued;
        EDX ← Processor Brand String, continued;
    BREAK;
    EAX = 80000005H:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    EAX = 80000006H:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Cache information;
        EDX ← Reserved = 0;
    BREAK;
    EAX = 80000007H:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    EAX = 80000008H:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    DEFAULT: (* EAX = Value outside of recognized range for CPUID. *)
        (* If the highest basic information leaf data depend on ECX input value, ECX is honored.*)
        EAX ← Reserved; (* Information returned for highest basic information leaf. *)
        EBX ← Reserved; (* Information returned for highest basic information leaf. *)
        ECX ← Reserved; (* Information returned for highest basic information leaf. *)
        EDX ← Reserved; (* Information returned for highest basic information leaf. *)
    BREAK;
ESAC;
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

In earlier IA-32 processors that do not support the CPUID instruction, execution of the instruction results in an invalid opcode (#UD) exception being generated.§

CHAPTER 3
SYSTEM PROGRAMMING FOR INTEL® AVX-512

This chapter describes the operating system programming considerations for supporting the following extended processor states: 512-bit ZMM registers and opmask k-registers. These system programming requirements apply to AVX-512 Foundation instructions and other 512-bit instructions described in Chapter 7.

The basic requirements for an operating system using XSAVE/XRSTOR to manage processor extended states, e.g. YMM registers, can be found in Chapter 13 of *Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A*. This chapter covers additional requirements for OS to support ZMM and opmask register states.

3.1 AVX-512 STATE, EVEX PREFIX AND SUPPORTED OPERATING MODES

AVX-512 instructions are encoded using EVEX prefix. The EVEX encoding scheme can support 512-bit, 256-bit and 128-bit instructions that operate on opmask register, ZMM, YMM and XMM states.

For processors that support AVX-512 family of instructions, the extended processor states (ZMM and opmask registers) exist in all operating modes. However, the access to those states may vary in different modes. The processor's support for instruction extensions that employ EVEX prefix encoding is independent of the processor's support for using XSAVE/XRSTOR/XSAVEOPT to those states.

Instructions requiring EVEX prefix encoding generally are supported in 64-bit, 32-bit modes, and 16-bit protected mode. They are not supported in Real mode, Virtual-8086 mode or entering into SMM mode.

Note that bits MAX_VL-1:256 (511:256) of ZMM register state are maintained across transitions into and out of these modes. Because the XSAVE/XRSTOR/XSAVEOPT instruction can operate in all operating modes, it is possible that the processor's ZMM register state can be modified by software in any operating mode by executing XRSTOR. The ZMM registers can be updated by XRSTOR using the state information stored in the XSAVE/XRSTOR area residing in memory.

3.2 AVX-512 STATE MANAGEMENT

Operating systems must use the XSAVE/XRSTOR/XSAVEOPT instructions for ZMM and opmask state management. An OS must enable its ZMM and opmask state management to support AVX-512 Foundation instructions. Otherwise, an attempt to execute an instruction in AVX-512 Foundation instructions (including a scalar 128-bit SIMD instructions using EVEX encoding) will cause a #UD exception. An operating system, which enabled AVX-512 state to support AVX-512 Foundation instructions, is also sufficient to support the rest of AVX-512 family of instructions.

3.2.1 Detection of ZMM and Opmask State Support

Hardware support of the extended state components for executing AVX-512 Foundation instructions is queried through the main leaf of CPUID leaf function 0DH with index ECX = 0. Specifically, the return value in EDX:EAX of CPUID.(EAX=0DH, ECX=0) provides a 64-bit wide bit vector of hardware support of processor state components, beginning with bit 0 of EAX corresponding to x87 FPU state, CPUID.(EAX=0DH, ECX=0):EAX[1] corresponding to SSE state (XMM registers and MXCSR), CPUID.(EAX=0DH, ECX=0):EAX[2] corresponding to YMM states.

The ZMM and opmaks states consist of three additional components in the XSAVE/XRSTOR state save area:

- The opmask register state component represents eight 64-bit opmask registers. Processor support for this component state is indicated by CPUID.(EAX=0DH, ECX=0):EAX[5].
- The ZMM_Hi256 component represents the high 256 bits of the low 16 ZMM registers, i.e. ZMM0..15[511:256]. Processor support for this component state is indicated by CPUID.(EAX=0DH, ECX=0):EAX[6].
- The Hi16_ZMM component represents the full 512 bits of the high 16 ZMM registers, i.e. ZMM16..31[511:0]. Processor support for this component state is indicated by CPUID.(EAX=0DH, ECX=0):EAX[7].

Each component state has a corresponding enable it in the XCR0 register. Operating system must use XSETBV to set these three enable bits to enable AVX-512 Foundation instructions to be decoded. The location of bit vector representing the AVX-512 states, matching the layout of the XCR0 register, is provided in the following figure.

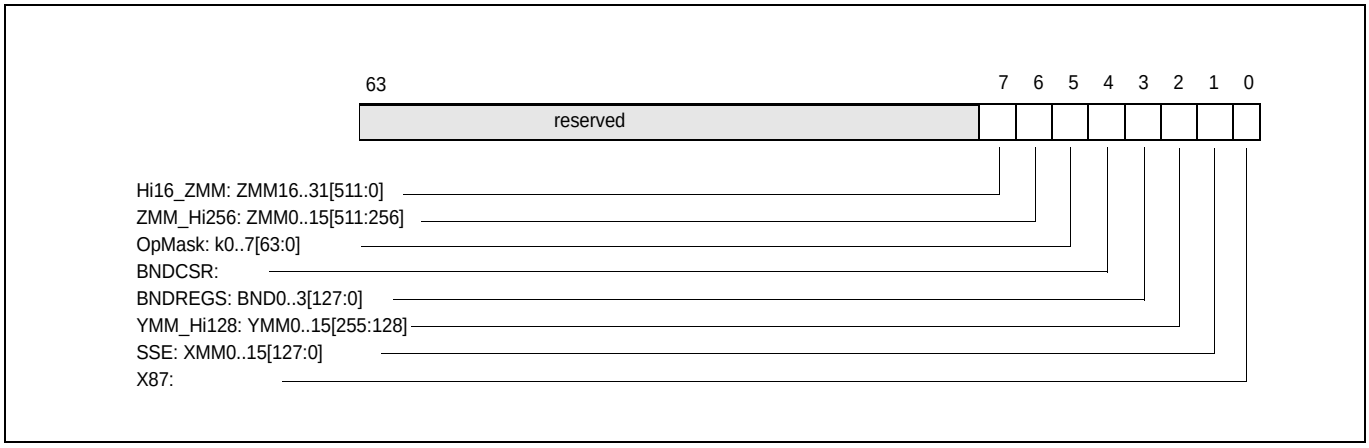


Figure 3-1. Bit Vector and XCR0 Layout of Extended Processor State Components

3.2.2 Enabling of ZMM and Opmask Register State

An OS can enable ZMM and opmask register state support with the following steps:

- Verify the processor supports XSAVE/XRSTOR/XSETBV/XGETBV instructions and the XCR0 register by checking CPUID.1.ECX.XSAVE[bit 26]=1.
- Verify the processor supports SSE, YMM, ZMM_Hi256, Hi16_ZMM, and opmask states (i.e. bits 2:1 and 7:5 of XCR0 are valid) by checking CPUID.(EAX=0DH, ECX=0):EAX[7:5].

  The OS must determine the buffer size requirement for the XSAVE area that will be used by XSAVE/XRSTOR. Note that even though ZMM8-ZMM31 are not accessible in 32 bit mode, a 32 bit OS is still required to allocate the buffer for the entire ZMM state.
- Set CR4.OSXSAVE[bit 18]=1 to enable the use of XSETBV/XGETBV instructions to write/read the XCR0 register.
- Supply an appropriate mask via EDX:EAX to execute XSETBV to enable the processor state components that the OS wishes to manage using XSAVE/XRSTOR instruction.

  To enable ZMM and opmask register state, system software must use a EDX:EAX mask of 111xx111b when executing XSETBV. Attempts to set XCR0[7:5] to any value other than 111b will result in a #GP.

Table 3-1. XCR0 Processor State Components

Bit	Meaning
0 - x87	This bit 0 must be 1. An attempt to write 0 to this bit causes a #GP exception.
1 - SSE	If 1, the processor supports SSE state (MXCSR and XMM registers) management using XSAVE, XSAVEOPT, and XRSTOR. This bit must be set to '1' to enable AVX-512 Foundation instructions.
2 - YMM_Hi128	If 1, the processor supports YMM_hi128 state management (upper 128 bits of YMM0-15) using XSAVE, XSAVEOPT, and XRSTOR. This bit must be set to '1' to enable AVX-512 Foundation instructions.
3 - BNDREGS	If 1, the processor supports Intel Memory Protection Extensions (Intel MPX) bound register state management using XSAVE, XSAVEOPT, and XRSTOR. See Section 9.3.2 for system programming requirement to enable Intel MPX.
4 - BNDCSR	If 1, the processor supports Intel MPX bound configuration and status management using XSAVE, XSAVEOPT, and XRSTOR. See Section 9.3.2 for system programming requirement to enable Intel MPX.

Table 3-1. XCR0 Processor State Components

Bit	Meaning
5 - Opmask	If 1, the processor supports the opmask state management using XSAVE, XSAVEOPT, and XRSTOR. This bit must be set to '1' to enable AVX-512 Foundation instructions.
6 - ZMM_Hi256	If 1, the processor supports ZMM_Hi256 state (the upper 256 bits of the low 16 ZMM registers) management using XSAVE, XSAVEOPT, and XRSTOR. This bit must be set to '1' to enable AVX-512 Foundation instructions.
7 - Hi16_ZMM	If 1, the processor supports Hi16-_ZMM state (the full 512 bits of the high16 ZMM registers) management using XSAVE, XSAVEOPT, and XRSTOR. This bit must be set to '1' to enable AVX-512 Foundation instructions.

3.2.3 Enabling of SIMD Floating-Exception Support

AVX-512 Foundation instructions may generate SIMD floating-point exceptions. An OS must enable SIMD floating-point exception support by setting CR4.OSXMMEXCPT[bit 10]=1.

The effect of CR4 setting that affects AVX-512 Foundation instructions is the same as for AVX and FMA enabling as listed in Table 3-2

Table 3-2. CR4 Bits for AVX-512 Foundation Instructions Technology Support

Bit	Meaning
CR4.OSXSAVE[bit 18]	If set, the OS supports use of XSETBV/XGETBV instruction to access. the XCR0 register, XSAVE/XRSTOR to manage processor extended states. Must be set to '1' to enable AVX-512 Foundation, AVX2, FMA, and AVX instructions.
CR4.OSXMMEXCPT[bit 10]	Must be set to 1 to enable SIMD floating-point exceptions. This applies to SIMD floating-point instructions across AVX-512 Foundation, AVX and FMA, and legacy 128-bit SIMD floating-point instructions operating on XMM registers.
CR4.OSFXSR[bit 9]	Must be set to 1 to enable legacy 128-bit SIMD instructions operating on XMM state. Not needed to enable AVX-512 Foundation, AVX2, FMA, and AVX instructions.

3.2.4 The Layout of XSAVE Sate Save Area

The OS must determine the buffer size requirement by querying CPUID with EAX=0DH, ECX=0. If the OS wishes to enable all processor extended state components in the XCR0, it can allocate the buffer size according to CPUID.(EAX=0DH, ECX=0):ECX.

After the memory buffer for XSAVE is allocated, the entire buffer must be cleared prior to executing XSAVE.

The XSAVE area layout currently defined in Intel Architecture is listed in Table 3-3. The register fields of the first 512 byte of the XSAVE area are identical to those of the FXSAVE/FXRSTOR area.

The layout of the XSAVE Area for additional processor components (512-bit ZMM register, 32 ZMM registers, opmask registers) are to be determined later.

Table 3-3. Layout of XSAVE Area For Processor Supporting YMM State

Save Areas	Offset (Byte)	Size (Bytes)
FPU/SSE SaveArea	0	512
Header	512	64
Ext_Save_Area_2 (YMM_Hi128)	CPUID.(EAX=0DH, ECX=2):EBX	CPUID.(EAX=0DH, ECX=2):EAX
Ext_Save_Area_3 (BNDREGS)	CPUID.(EAX=0DH, ECX=3):EBX	CPUID.(EAX=0DH, ECX=3):EAX
Ext_Save_Area_4 (BNDCSR)	CPUID.(EAX=0DH, ECX=4):EBX	CPUID.(EAX=0DH, ECX=4):EAX
Ext_Save_Area_5 (OPMASK)	CPUID.(EAX=0DH, ECX=5):EBX	CPUID.(EAX=0DH, ECX=5):EAX
Ext_Save_Area_6 (ZMM_Hi256)	CPUID.(EAX=0DH, ECX=6):EBX	CPUID.(EAX=0DH, ECX=6):EAX
Ext_Save_Area_7 (Hi16_ZMM)	CPUID.(EAX=0DH, ECX=7):EBX	CPUID.(EAX=0DH, ECX=7):EAX

The format of the header is as follows (see Table 3-4):

Table 3-4. XSAVE Header Format

15:8	7:0	Byte Offset from Header	Byte Offset from XSAVE Area
Reserved (Must be zero)	XSTATE_BV	0	512
Reserved	Reserved (Must be zero)	16	528
Reserved	Reserved	32	544
Reserved	Reserved	48	560

The layout of the Ext_Save_Area[YMM_Hi128] contains 16 of the upper 128-bits of the YMM registers, it is shown in Table 3-5.

Table 3-5. XSAVE Save Area Layout for YMM_Hi128 State (Ext_Save_Area_2)

31 16	15 0	Byte Offset from YMM_Hi128_Save_Area	Byte Offset from XSAVE Area
YMM1[255:128]	YMM0[255:128]	0	576
YMM3[255:128]	YMM2[255:128]	32	608
YMM5[255:128]	YMM4[255:128]	64	640
YMM7[255:128]	YMM6[255:128]	96	672
YMM9[255:128]	YMM8[255:128]	128	704
YMM11[255:128]	YMM10[255:128]	160	736
YMM13[255:128]	YMM12[255:128]	192	768
YMM15[255:128]	YMM14[255:128]	224	800

The layout of the Ext_SAVE_Area_3[BNDREGS] contains bounds register state of the Intel Memory Protection Extensions (Intel MPX), which is described in Section 9.3.2.

The layout of the Ext_SAVE_Area_4[BNDCSR] contains the processor state of bounds configuration and status of Intel MPX, which is described in Section 9.3.2.

The layout of the Ext_SAVE_Area_5[Opmask] contains 8 64-bit mask register as shown in Table 3-6.

Table 3-6. XSAVE Save Area Layout for Opmask Registers

15 8	7 0	Byte Offset from OPMASK_Save_Area	Byte Offset from XSAVE Area
K1[63:0]	K0[63:0]	0	1088
K3[63:0]	K2[63:0]	16	1104
K5[63:0]	K4[63:0]	32	1120
K7[63:0]	K6[63:0]	48	1136

The layout of the Ext_SAVE_Area_6[ZMM_Hi256] is shown below in Table 3-7.

Table 3-7. XSAVE Save Area Layout for ZMM State of the High 256 Bits of ZMM0-ZMM15 Registers

63 32	31 0	Byte Offset from ZMM_Hi256_Save_Area	Byte Offset from XSAVE Area
ZMM1[511:256]	ZMM0[511:256]	0	1152
ZMM3[511:256]	ZMM2[511:256]	64	1216
ZMM5[511:256]	ZMM4[511:256]	128	1280
ZMM7[511:256]	ZMM6[511:256]	192	1344
ZMM9[511:256]	ZMM8[511:256]	256	1408
ZMM11[511:256]	ZMM10[511:256]	320	1472
ZMM13[511:256]	ZMM12[511:256]	384	1536
ZMM15[511:256]	ZMM14[511:256]	448	1600

The layout of the Ext_SAVE_Area_7[Hi16_ZMM] corresponding to the upper new 16 ZMM registers is shown below in Table 3-8.

Table 3-8. XSAVE Save Area Layout for ZMM State of ZMM16-ZMM31 Registers

127 64	63 0	Byte Offset from Hi16_ZMM_Save_Area	Byte Offset from XSAVE Area
ZMM17[511:0]	ZMM16[511:0]	0	1664
ZMM19[511:0]	ZMM18[511:0]	128	1792
ZMM21[511:0]	ZMM20[511:0]	256	1920
ZMM23[511:0]	ZMM22[511:0]	384	2048
ZMM25[511:0]	ZMM24[511:0]	512	2176
ZMM27[511:0]	ZMM26[511:0]	640	2304
ZMM29[511:0]	ZMM28[511:0]	768	2432
ZMM31[511:0]	ZMM30[511:0]	896	2560

3.2.5 XSAVE/XRESTOR Interaction with YMM State and MXCSR

The processor's actions as a result of executing XRSTOR, on the MXCSR, XMM and YMM registers, are listed in Table 3-9 The XMM registers may be initialized by the processor (See XRSTOR operation in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*). When the MXCSR register is updated from memory, reserved bit checking is enforced. XSAVE / XRSTOR will save / restore the MXCSR only if the AVX or SSE bits are set in the EDX:EAX mask.

Table 3-9. XRSTOR Action on MXCSR, XMM Registers, YMM Registers

EDX:EAX		XSTATE_BV		MXCSR	YMM_Hi128 Registers	XMM Registers
Bit 2	Bit 1	Bit 2	Bit 1			
0	0	X	X	None	None	None
0	1	X	0	Load/Check	None	Init by processor
0	1	X	1	Load/Check	None	Load
1	0	0	X	Load/Check	Init by processor	None
1	0	1	X	Load/Check	Load	None
1	1	0	0	Load/Check	Init by processor	Init by processor
1	1	0	1	Load/Check	Init by processor	Load
1	1	1	0	Load/Check	Load	Init by processor
1	1	1	1	Load/Check	Load	Load

The action of XSAVE for managing YMM and MXCSR is listed in Table 3-10.

Table 3-10. XSAVE Action on MXCSR, XMM, YMM Register

EDX:EAX		XCR0_MASK		MXCSR	YMM_H Registers	XMM Registers
Bit 2	Bit 1	Bit 2	Bit 1			
0	0	X	X	None	None	None
0	1	X	1	Store	None	Store
0	1	X	0	None	None	None
1	0	0	X	None	None	None
1	0	1	1	Store	Store	None
1	1	0	0	None	None	None
1	1	0	1	Store	None	Store
1	1	1	1	Store	Store	Store

3.2.6 XSAVE/XRESTOR/XSAVEOPT and Managing ZMM and Opmask States

The requirements for managing ZMM_Hi256, Hi16_ZMM and Opmask registers using XSAVE/XRSTOR/XSAVEOPT are simpler than those listed in Section 3.2.5. Because each of the three components (ZMM_Hi256, Hi16_ZMM and Opmask registers) can be managed independently of one another by XSAVE/XRSTOR/XSAVEOPT according to the corresponding bits in the bit vectors: EDX:EAX, XSAVE_BV, XCR0_MASK, independent of MXSCR:

- For using XSAVE with Opmask/ZMM_Hi256/Hi16_ZMM, XSAVE/XSAVEOPT will save the component to memory and mark the corresponding bits in the XSTATE_BV of the XSAVE header, if that component is specified in EDX:EAX as input to XSAVE/XSAVEOPT.
- XRSTOR will restore the Opmask/ZMM_Hi256/Hi16_ZMM components by checking the corresponding bits in both the input bit vector in EDX:EAX of XRSTOR and in XSTATE_BV of the header area in the following ways:
  - If the corresponding bit in EDX:EAX is set and XSTATE_BV is INIT, that component will be initialized,
  - If the corresponding bit in EDX:EAX is set and XSTATE_BV is set, that component will be restored from memory,
  - If the corresponding bit in EDX:EAX is not set, that component will remain unchanged.
- To enable AVX-512 Foundation instructions, all three components (Opmask/ZMM_Hi256/Hi16_ZMM) in XCR0 must be set.

The processor supplied INIT values for each processor state component used by XRSTOR is listed in Table 3-11.

Table 3-11. Processor Supplied Init Values XRSTOR May Use

Processor State Component	Processor Supplied Register Values
x87 FPU State	FCW ← 037FH; FTW ← 0FFFFH; FSW ← 0H; FPU CS ← 0H; FPU DS ← 0H; FPU IP ← 0H; FPU DP ← 0; ST0-ST7 ← 0;
SSE State[1]	If 64-bit Mode: XMM0-XMM15 ← 0H; Else XMM0-XMM7 ← 0H
YMM_Hi128 State[1]	If 64-bit Mode: YMM0_H-YMM15_H ← 0H; Else YMM0_H-YMM7_H ← 0H
OPMASK State[1]	If 64-bit Mode: K0-K7 ← 0H;
ZMM_Hi256 State[1]	If 64-bit Mode: ZMM0_H-ZMM15_H ← 0H; Else ZMM0_H-ZMM7_H ← 0H
Hi16_ZMM State[1]	If 64-bit Mode: ZMM16-ZMM31 ← 0H;

NOTES:

1. MXCSR state is not updated by processor supplied values. MXCSR state can only be updated by XRSTOR from state information stored in XSAVE/XRSTOR area.

3.3 RESET BEHAVIOR

At processor reset

- YMM0-15 bits[255:0] are set to zero.
- ZMM0-15 bits [511:256] are set to zero.
- ZMM16-31 are set to zero.
- Opmask register K0-7 are set to 0x0H.
- **XCR0**[2:1] is set to zero, **XCR0**[0] is set to 1.
- **XCR0**[7:6] and is set to zero, **XCR0**[Opmask] is set to 0.
- CR4.OSXSAVE[bit 18] (and its mirror CPUID.1.ECX.OSXSAVE[bit 27]) is set to 0.

3.4 EMULATION

Setting the CR0.EM bit to 1 provides a technique to emulate Legacy SSE floating-point instruction sets in software. This technique is not supported with AVX instructions, nor FMA instructions.

If an operating system wishes to emulate AVX instructions, set XCR0[2:1] to zero. This will cause AVX instructions to #UD. Emulation of FMA by operating system can be done similarly as with emulating AVX instructions.

3.5 WRITING FLOATING-POINT EXCEPTION HANDLERS

AVX-512, AVX and FMA floating-point exceptions are handled in an entirely analogous way to Legacy SSE floating-point exceptions. To handle unmasked SIMD floating-point exceptions, the operating system or executive must provide an exception handler. The section titled "SSE and SSE2 SIMD Floating-Point Exceptions" in Chapter 11, "Programming with Streaming SIMD Extensions 2 (SSE2)," of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, describes the SIMD floating-point exception classes and gives suggestions for writing an exception handler to handle them.

To indicate that the operating system provides a handler for SIMD floating-point exceptions (#XM), the CR4.OSXMMEXCPT flag (bit 10) must be set.

This page was intentionally left blank.

CHAPTER 4
AVX-512 INSTRUCTION ENCODING

4.1 OVERVIEW SECTION

This chapter describes the details of AVX-512 instruction encoding system. The AVX-512 Foundation instruction described in Chapter 5 use a new prefix (called EVEX). Opmask instructions described in Chapter 6 are encoded using the VEX prefix. The EVEX prefix has some parts resembling the instruction encoding scheme using the VEX prefix, and many other capabilities not available with the VEX prefix. The EVEX encoding architecture also applies to other 512-bit instructions described in Chapter 7.

The significant feature differences between EVEX and VEX are summarized below.

- EVEX is a 4-Byte prefix (the first byte must be 62H); VEX is either a 2-Byte (C5H is the first byte) or 3-Byte (C4H is the first byte) prefix.
- EVEX prefix can encode 32 vector registers (XMM/YMM/ZMM) in 64-bit mode.
- EVEX prefix can encode an opmask register for conditional processing or selection control in EVEX-encoded vector instructions; opmask instructions, whose source/destination operands are opmask registers and treat the content of an opmask register as a single value, are encoded using the VEX prefix.
- EVEX memory addressing with disp8 form uses a compressed disp8 encoding scheme to improve encoding density of the instruction byte stream.
- EVEX prefix can encode functionality that are specific to instruction classes (e.g. packed instruction with "load+op" semantic can support embedded broadcast functionality, floating-point instruction with rounding semantic can support static rounding functionality, floating-point instruction with non-rounding arithmetic semantic can support "suppress all exceptions" functionality).

4.2 INSTRUCTION FORMAT AND EVEX

The placement of the EVEX prefix in an IA instruction is represented in Figure 4-1:

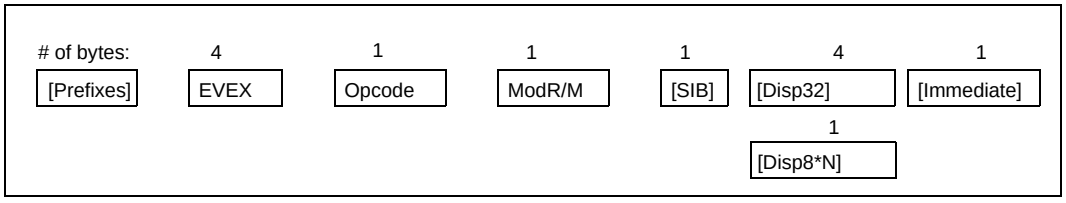


Figure 4-1. AVX-512 Instruction Format and the EVEX Prefix

The EVEX prefix is a 4-byte prefix, with the first two bytes derived from unused encoding form of the 32-bit-mode-only BOUND instruction. The layout of the EVEX prefix is shown in Figure 4-2. The first byte must be 62H, followed by three payload bytes, denoted as P0, P1, and P2 individually or collectively as P[23:0] (see Figure 4-2).

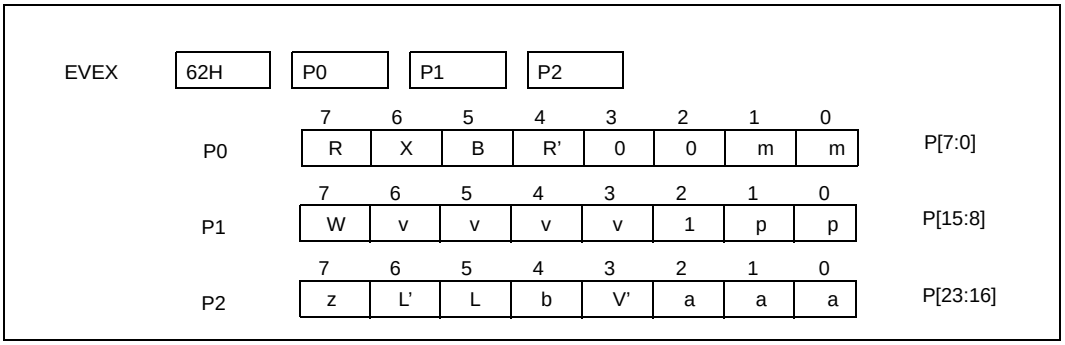


Figure 4-2. Bit Field Layout of the EVEX Prefix

Table 4-1. EVEX Prefix Bit Field Functional Grouping

Notation	Bit field Group	Position	Comment
--	Reserved	P[3 : 2]	Must be 0
--	Fixed Value	P[10]	Must be 1
EVEX.mm	Compressed legacy escape	P[1: 0]	Identical to low two bits of VEX.mmmmm
EVEX.pp	Compressed legacy prefix	P[9 : 8]	Identical to VEX.pp
EVEX.RXB	Next-8 register specifier modifier	P[7 : 5]	Combine with ModR/M.reg, ModR/M.rm (base, index/vidx)
EVEXR'	High-16 register specifier modifier	P[4]	Combine with EVEX.R and ModR/M.reg
EVEXX	High-16 register specifier modifier	P[6]	Combine with EVEX.B and ModR/M.rm, when SIB/VSIB absent
EVEX.vvvv	NDS register specifier	P[14 : 11]	Same as VEX.vvvv
EVEXV'	High-16 NDS/VIDX register specifier	P[19]	Combine with EVEX.vvvv or when VSIB present
EVEX.aaa	Embedded opmask register specifier	P[18 : 16]	
EVEX.W	Osize promotion/Opcode extension	P[15]	
EVEX.z	Zeroing/Merging	P[23]	
EVEX.b	Broadcast/RC/SAE Context	P[20]	
EVEX.L'L	Vector length/RC	P[22 : 21]	

The bit fields in P[23:0] are divided into the following functional groups (Table 4-1 provides a tabular summary):

- Reserved bits: P[3:2] must be 0, otherwise #UD.
- Fixed-value bit: P[10] must be 1, otherwise #UD,
- Compressed legacy prefix/escape bytes: P[1:0] is identical to the lowest 2 bits of VEX.mmmmm; P[9:8] is identical to VEX.pp.
- Operand specifier modifier bits for vector register, general purpose register, memory addressing: P[7:5] allows access to the next set of 8 registers beyond the low 8 registers when combined with ModR/M register specifiers.
- Operand specifier modifier bit for vector register: P[4] (or EVEX.R') allows access to the high 16 vector register set when combined with P[7] and ModR/M.reg specifier; P[6] can also provide access to a high 16 vector register when SIB or VSIB addressing are not needed.
- Non-destructive source /vector index operand specifier: P[19] and P[14:11] encode the second source vector register operand in a non-destructive source syntax, vector index register operand can access an upper 16 vector register using P[19]
- Op-mask register specifiers: P[18:16] encodes op-mask register set k0-k7 in instructions operating on vector registers,

- EVEX.W: P[15] is similar to VEX.W which serves either as opcode extension bit or operand size promotion to 64-bit in 64-bit mode,
- Vector destination merging/zeroing: P[23] encodes the destination result behavior which either zeroes the masked elements or leave masked element unchanged,
- Broadcast/Static-rounding/SAE context bit: P[20] encodes multiple functionality, which differs across different classes of instructions and can affect the meaning of the remaining field (EVEX.L'L). The functionality for the following instruction classes are:
  - broadcasting a single element across the destination vector register: this applies to the instruction class with Load+Op semantic where one of the source operand is from memory.
  - Redirect L'L field (P[22:21]) as static rounding control for floating-point instructions with rounding semantic. Static rounding control overrides MXCSR.RC field and implies "Suppress all exceptions" (SAE).
  - Enable SAE for floating -point instructions with arithmetic semantic that is not rounding.
  - For instruction classes outside of the afore-mentioned three classes, setting EVEX.b will cause #UD.
- Vector length/rounding control specifier: P[22:21] can server one of three functionality:
  - vector length information for packed vector instructions,
  - ignored for instructions operating on vector register content as a single data element,
  - rounding control for floating-point instructions that have a rounding semantic and whose source and destination operands are all vector registers.

4.3 REGISTER SPECIFIER ENCODING AND EVEX

EVEX-encoded instruction can access 8 opmask registers, 16 general-purpose registers and 32 vector registers in 64-bit mode (8 general-purpose registers and 8 vector registers in non-64-bit modes). EVEX-encoding can support instruction syntax that access up to 4 instruction operands. Normal memory addressing modes and VSIB memory addressing are supported with EVEX prefix encoding. The mapping of register operands used by various instruction syntax and memory addressing in 64-bit mode are shown in Table 4-2. Opmask register encoding is described in Section 4.3.1.

Table 4-2. 32-Register Support in 64-bit Mode Using EVEX with Embedded REX Bits

	4[1]	3	[2:0]	Reg. Type	Common Usages
REG	EVEX.R'	REXR	modrm.reg	GPR, Vector	Destination or Source
NDS/NDD	EVEX.V'	EVEX.vvvv		GPR, Vector	2ndSource or Destination
RM	EVEXX	EVEXB	modrm.r/m	GPR, Vector	!st Source or Destination
BASE	0	EVEXB	modrm.r/m	GPR	memory addressing
INDEX	0	EVEXX	sib.index	GPR	memory addressing
VIDX	EVEX.V'	EVEXX	sib.index	Vector	VSIB memory addressing
IS4	Imm8[3]	Imm8[7:4]		Vector	3rd Source

NOTES:

1. Not applicable for accessing general purpose registers.

The mapping of register operands used by various instruction syntax and memory addressing in 32-bit modes are shown in Table 4-3.

Table 4-3. EVEX Encoding Register Specifiers in 32-bit Mode

	[2:0]	Reg. Type	Common Usages
REG	modrm.reg	GPR, Vector	Dest or Source

Table 4-3. EVEX Encoding Register Specifiers in 32-bit Mode

NDS/NDD	EVEX.vvv	GPR, Vector	2ndSource or Dest
RM	modrm.r/m	GPR, Vector	!st Source or Dest
BASE	modrm.r/m	GPR	memory addressing
INDEX	sib.index	GPR	memory addressing
VIDX	sib.index	Vector	VSIB memory addressing
IS4	Imm8[7:5]	Vector	3rd Source

4.3.1 Opmask Register Encoding

There are eight opmask registers, k0-k7. Opmask register encoding falls into two categories:

- Opmask registers that are the source or destination operands of an instruction treating the content of opmask register as a scalar value, are encoded using the VEX prefix scheme. It can support up to three operands using standard modR/M byte's reg field and rm field and VEX.vvvv. Such a scalar opmask instruction does not support conditional update of the destination operand.
- An opmask register providing conditional processing and/or conditional update of the destination register of a vector instruction is encoded using EVEX.aaa field (see Section 4.4).
- An opmask register serving as the destination or source operand of a vector instruction is encoded using standard modR/M byte's reg field and rm fields.

Table 4-4. Opmask Register Specifier Encoding

	[2:0]	Register Access	Common Usages
REG	modrm.reg	k0-k7	Source
NDS	VEX.vvv	k0-k7	2ndSource
RM	modrm.r/m	k0-7	!st Source
{k1}	EVEX.aaa	k0[1]-k7	Opmask

NOTES:

1. instructions that overwrite the conditional mask in opmask do not permit using k0 as the embedded mask.

4.4 MASKING SUPPORT IN EVEX

EVEX can encode an opmask register to conditionally control per-element computational operation and updating of result of an instruction to the destination operand. The predicate operand is known as the opmask register. The EVEX.aaa field, P[18:16] of the EVEX prefix, is used to encode one out of a set of eight 64-bit architectural registers. Note that from this set of 8 architectural registers, only k1 through k7 can be addressed as predicate operands. k0 can be used as a regular source or destination but cannot be encoded as a predicate operand.

AVX-512 instructions support two types of masking with EVEX.z bit (P[23]) controlling the type of masking:

- Merging-masking, which is the default type of masking for EVEX-encoded vector instructions, preserves the old value of each element of the destination where the corresponding mask bit has a 0. It corresponds to the case of EVEX.z = 0.
- Zeroing-masking, is enabled by having the EVEX.z bit set to 1. In this case, an element of the destination is set to 0 when the corresponding mask bit has a 0 value.

AVX-512 Foundation instructions can be divided in three different groups:

- Instructions which support "zeroing-masking".
    - Also allow merging-masking.

- Instructions which require aaa = 000.
  - Do not allow any form of masking.
- Instructions which allow merging-masking but do not allow zeroing-masking
  - Require EVEX.z to be set to 0
  - This group is mostly composed of instructions that write to memory.
- Instructions which require aaa <> 000 do not allow EVEX.z to be set to 1.
  - Allow merging-masking and do not allow zeroing-masking, e.g., gather instructions.

4.5 COMPRESSED DISPLACEMENT (DISP8*N) SUPPORT IN EVEX

For memory addressing using disp8 form, EVEX-encoded instructions always use a compressed displacement scheme by multiplying disp8 in conjunction with a scaling factor N that is determined based on the vector length, the value of EVEX.b bit (embedded broadcast) and the input element size of the instruction. In general, the factor N corresponds to the number of bytes characterizing the internal memory operation of the input operand (e.g., 64 when the accessing a full 512-bit memory vector). The scale factor N is listed in Table 4-5 and Table 4-6 below, where EVEX encoded instructions are classified using the **tupletype** attribute. The scale factor N of each tupletype is listed based on the vector length (VL) and other factors affecting it.

Table 4-5 covers EVEX-encoded instructions which has a load semantic in conjunction with additional computational or data element movement operation, operating either on the full vector or half vector (due to conversion of numerical precision from a wider format to narrower format). EVEX.b is supported for such instructions for data element sizes which are either dword or qword (see Section 4.7).

EVEX-encoded instruction that are pure load/store, and "Load+op" instruction semantic that operate on data element size less then dword do not support broadcasting using EVEX.b. These are listed in Table 4-6. Table 4-6 also includes many broadcast instructions which perform broadcast using a subset of data elements without using EVEX.b. These instructions and a few data element size conversion instruction are covered in Table 4-6. Instruction classified in Table 4-6 do not use EVEX.b and EVEX.b must be 0, otherwise #UD will occur.

The tupletype abbreviation will be referenced in the instruction operand encoding table in the reference page of each instruction, providing the cross reference for the scaling factor N to encoding memory addressing operand.

Note that the disp8*N rules still apply when using 16b addressing.

Table 4-5. Compressed Displacement (DISP8*N) Affected by Embedded Broadcast

TupleType	EVEX.b	InputSize	EVEX.W	Broadcast	N (VL=128)	N (VL=256)	N (VL= 512)	Comment
Full Vector (FV)	0	32bit	0	none	16	32	64	Load+Op (Full Vector Dword/Qword)
	1	32bit	0	{1tox}	4	4	4	
	0	64bit	1	none	16	32	64	
	1	64bit	1	{1tox}	8	8	8	
Half Vector (HV)	0	32bit	0	none	8	16	32	Load+Op (Half Vector)
	1	32bit	0	{1tox}	4	4	4	

Table 4-6. EVEX DISP8*N For Instructions Not Affected by Embedded Broadcast

TupleType	InputSize	EVEX.W	N (VL= 128)	N (VL= 256)	N (VL= 512)	Comment
Full Vector Mem (FVM)	N/A	N/A	16	32	64	Load/store or subDword full vector

Table 4-6. EVEX DISP8*N For Instructions Not Affected by Embedded Broadcast (Contd.)

TupleType	InputSize	EVEX.W	N (VL= 128)	N (VL= 256)	N (VL= 512)	Comment
Tuple1 Scalar (T1S)	8bit	N/A	1	1	1	1Tuple less than Full Vector
	16bit	N/A	2	2	2	
	32bit	0	4	4	4	
	64bit	1	8	8	8	
Tuple1 Fixed (T1F)	32bit	N/A	4	4	4	1 Tuple memsize not affected by EVEX.W
	64bit	N/A	8	8	8	
Tuple2 (T2)	32bit	0	8	8	8	Broadcast (2 elements)
	64bit	1	NA	16	16	
Tuple4 (T4)	32bit	0	NA	16	16	Broadcast (4 elements)
	64bit	1	NA	NA	32	
Tuple8 (T8)	32bit	0	NA	NA	32	Broadcast (8 elements)
Half Mem (HVM)	N/A	N/A	8	16	32	SubQword Conversion
QuarterMem (QVM)	N/A	N/A	4	8	16	SubDword Conversion
OctMem (OVM)	N/A	N/A	2	4	8	SubWord Conversion
Mem128 M128)	N/A	N/A	16	16	16	Shift count from memory
MOVDDUP (DUP)	N/A	N/A	8	32	64	VMOVDDUP

4.6 EVEX ENCODING OF BROADCAST/ROUNDING/SAE SUPPORT

EVEX.b can provide three types of encoding context, depending on the instruction classes:

- Embedded broadcasting of one data element from a source memory operand to the destination for vector instructions with "load+op" semantic.
- Static rounding control overriding MXCSR.RC for floating-point instructions with rounding semantic.
- "Suppress All exceptions" (SAE) overriding MXCSR mask control for floating-point arithmetic instructions that does not have rounding semantic.

4.6.1 Embedded Broadcast Support in EVEX

EVEX encodes an embedded broadcast functionality that is supported on many vector instructions with 32-bit (double word or single-precision floating-point) and 64-bit data elements, and when the source operand is from memory. EVEX.b (P[20]) bit is used to enable broadcast on load-op instructions. When enabled, only one element is loaded from memory and broadcasted to all other elements instead of loading the full memory size.

The following instruction classes do not support embedded broadcasting:

- Instructions with only one scalar result is written to the vector destination.
- Instructions with explicit broadcast functionality provided by its opcode.
- Instruction semantic is a pure load or a pure store operation.

4.6.2 Static Rounding Support in EVEX

Static rounding control embedded in the EVEX encoding system applies only to register-to-register flavor of floating-point instructions with rounding semantic at two distinct vector lengths: (i) scalar, (ii) 512-bit. In both cases, the field EVEX.L'L expresses rounding mode control overriding MXCSR.RC if EVEX.b is set. When EVEX.b is set, "suppress all exceptions" is implied. The processor behave as if all MXCSR masking controls are set.

4.6.3 SAE Support in EVEX

The EVEX encoding system allows arithmetic floating-point instructions without rounding semantic to be encoded with the SAE attribute. This capability applies to scalar and all vector lengths, by setting EVEX.b. When EVEX.b is set, "suppress all exceptions" is implied. The processor behave as if all MXCSR masking controls are set.

4.6.4 Vector Length Orthogonality

The architecture of EVEX encoding scheme can support SIMD instructions operating at multiple vector lengths. Many AVX-512 Foundation instructions operate at 512-bit vector length. The vector length of EVEX encoded vector instructions are generally determined using the L'L field in EVEX prefix, except for 512-bit floating-point, reg-reg instructions with rounding semantic. The table below shows the vector length corresponding to various values of the L'L bits. When EVEX is used to encode scalar instructions, L'L is generally ignored.

When EVEX.b bit is set for a register-register instructions with floating-point rounding semantic, the same two bits P2[6:5] specifies rounding mode for the instruction, with implied SAE behavior. The mapping of different instruction classes relative to the embedded broadcast/rounding/SAE control and the EVEX.L'L fields are summarized in Table 4-7.

Table 4-7. EVEX Embedded Broadcast/Rounding/SAE and Vector Length on Vector Instructions

Position	P2[4]	P2[6:5]	P2[6:5]
Broadcast/Rounding/SAE Context	**EVEX.b**	**EVEX.L'L**	**EVEX.RC**
Reg-reg, FP Instructions w/ rounding semantic	Enable static rounding control (SAE implied)	Vector length Implied (512 bit or scalar)	00b: SAE + RNE 01b: SAE + RD 10b: SAE + RU 11b: SAE + RZ
FP Instructions w/o rounding semantic, can cause #XF	SAE control	00b: 128-bit 01b: 256-bit 10b: 512-bit 11b: Reserved (#UD)	NA
Load+op Instructions w/ memory source	Broadcast Control		NA
Other Instructions (Explicit Load/Store/Broadcast/Gather/Scatter, ..)	Must be 0 (otherwise #UD)		NA

4.7 #UD EQUATIONS FOR EVEX

Instructions encoded using EVEX can face three types of UD conditions: state dependent, opcode independent and opcode dependent.

4.7.1 State Dependent #UD

In general, attempts of execute an instruction, which required OS support for incremental extended state component, will #UD if required state components were not enabled by OS. Table 4-8 lists instruction categories with respect to required processor state components. Attempts to execute a given category of instructions while enabled states were less than the required bit vector in XCR0 shown in Table 4-8 will cause #UD.

Table 4-8. OS XSAVE Enabling Requirements of Instruction Categories

Instruction Categories	Vector Register State Access	Required XCR0 Bit Vector [7:0]
Legacy SIMD prefix encoded Instructions (e.g SSE)	XMM	xxxxxx11b
VEX-encoded instructions operating on YMM	YMM	xxxxx111b
EVEX-encoded 128-bit instructions	ZMM	111xx111b
EVEX-encoded 512-bit instructions	ZMM	111xx111b
VEX-encoded instructions operating on opmask	k-reg	xx1xxx11b

4.7.2 Opcode Independent #UD

A number of bit fields in EVEX encoded instruction must obey mode-specific but opcode-independent patterns listed in Table 4-9:

Table 4-9. Opcode Independent, State Dependent EVEX Bit Fields

Position	Notation	64-bit #UD	Non-64-bit #UD
P[3 : 2]	--	if > 0	if > 0
P[10]	--	if 0	if 0
P[1: 0]	EVEX.mm	if 00b	if 00b
P[7 : 6]	EVEX.RX	None (valid)	None (BOUND if EVEX.RX != 11b)

4.7.3 Opcode Dependent #UD

This section describes legal values for the rest of the EVEX bit fields. Table 4-10 lists the #UD conditions of EVEX prefix bit fields which encodes or modifies register operands.

Table 4-10. #UD Conditions of Operand-Encoding EVEX Prefix Bit Fields

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.R	P[7]	ModRM.reg encodes k-reg	if EVEX.R = 0	None (BOUND if EVEX.RX != 11b)
		ModRM.reg is opcode extension	None (ignored)	
		ModRM.reg encodes all other registers	None (valid)	
EVEX.X	P[6]	ModRM.r/m encodes ZMM/YMM/XMM	None (valid)	
		ModRM.r/m encodes k-reg or GPR	None (ignored)	
		ModRM.r/m without SIB/VSIB	None (ignored)	
		ModRM.r/m with SIB/VSIB	None (valid)	
EVEX.B	P[5]	ModRM.r/m encodes k-reg	None (ignored)	None (ignored)
		ModRM.r/m encodes other registers	None (valid)	
		ModRM.r/m base present	None (valid)	
		ModRM.r/m base not present	None (ignored)	
EVEXR'	P[4]	ModRM.reg encodes k-reg or GPR	if 0	None (ignored)
		ModRM.reg is opcode extension	None (ignored)	
		ModRM.reg encodes ZMM/YMM/XMM	None (valid)	
EVEX.vvvv	P[14 : 11]	vvvv encodes ZMM/YMM/XMM	None (valid)	if P[14] = 0
		otherwise	if != 1111b	if != 1111b
EVEXV'	P[19]	encodes ZMM/YMM/XMM	None (valid)	if 0
		otherwise	if 0	if 0

Table 4-11 lists the #UD conditions of instruction encoding of opmask register using EVEX.aaa or VEX.vvv

Table 4-11. #UD Conditions of Opmask Related Encoding Field

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
VEX.vvvv	varies	k-regs are instruction operands not mask control	if vvvv = 0xxxb	if vvvv = 0xxxb

Table 4-11. #UD Conditions of Opmask Related Encoding Field (Contd.)

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.aaa	P[18 : 16]	instructions do not use opmask for conditional processing[1]	if aaa != 000b	if aaa != 000b
		opmask used as conditional processing mask and updated at completion[2]	if aaa = 000b	if aaa = 000b;
		opmask used as conditional processing	None (valid[3])	None (valid[1])
EVEX.z	P[23]	vector instruction using opmask as source or destination	if EVEX.z != 0	if EVEX.z != 0
		store instructions or gather/scatter instructions	if EVEX.z != 0	if EVEX.z != 0
		instruction supporting conditional processing mask with EVEX.aaa = 000b	if EVEX.z != 0	if EVEX.z != 0

NOTES:

1. E.g. VBROADCASTMxxx
2. E.g. Gather/Scatter family
3. aaa can take any value. A value of 000 indicates that there is no masking on the instruction; in this case, all elements will be processed as if there was a mask of 'all ones' regardless of the actual value in K0.

Table 4-12 lists the #UD conditions of EVEX bit fields that depends on the context of EVEX.b.

Table 4-12. #UD Conditions Dependent on EVEX.b Context

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.L'Lb	P[22 : 20]	reg-reg, FP instructions with rounding semantic	None (valid[1])	None (valid[1])
		other reg-reg, FP instructions that can cause #XF	None (valid[2])	None (valid[2])
		other reg-mem instructions in Table 4-5	None (valid[3])	None (valid[3])
		other instruction classes[4] in Table 4-6	if EVEX.b > 0	if EVEX.b > 0

NOTES:

1. L'L specifies rounding control, see Table 4-7, supports {er} syntax.
2. L'L specifies vector length, see Table 4-7, supports {sae} syntax.
3. L'L specifies vector length, see Table 4-7, supports embedded broadcast syntax
4. L'L specifies either vector length or ignored.

4.8 DEVICE NOT AVAILABLE

EVEX-encoded instructions follow the same rules when it comes to generating #NM (Device Not Available) exception. In particular, it is generated when CR0.TS[bit 3]= 1.

4.9 SCALAR INSTRUCTIONS

EVEX-encoded scalar SIMD instructions can access up to 32 registers in 64-bit mode. Scalar instructions support masking (using the least significant bit of the opmask register), but broadcasting is not supported.

4.10 EXCEPTION CLASSIFICATIONS OF EVEX-ENCODED INSTRUCTIONS

The exception behavior of EVEX-encoded instructions can be classified into the classes shown in the rest of this section. The classification of EVEX-encoded instructions follow a similar framework as those of AVX and AVX2 instructions using the VEX prefix. Exception types for EVEX-encoded instructions are named in the style of

"E##" or with a suffix "E##XX". The "##" designation generally follows that of AVX/AVX2 instructions. The majority of EVEX encoded instruction with "Load+op" semantic supports memory fault suppression, which is represented by E##. The instructions with "Load+op" semantic but do not support fault suppression are named "E##NF". A summary table of exception classes by class names are shown below.

Table 4-13. EVEX-Encoded Instruction Exception Class Summary

Exception Class	Instruction set	Mem arg	(#XM)
Type E1	Vector Moves/Load/Stores	explicitly aligned, w/ fault suppression	none
Type E1NF	Vector Non-temporal Stores	explicitly aligned, no fault suppression	none
Type E2	FP Vector Load+op	Support fault suppression	yes
Type E2NF	FP Vector Load+op	No fault suppression	yes
Type E3	FP Scalar/Partial Vector, Load+Op	Support fault suppression	yes
Type E3NF	FP Scalar/Partial Vector, Load+Op	No fault suppression	yes
Type E4	Integer Vector Load+op	Support fault suppression	no
Type E4NF	Integer Vector Load+op	No fault suppression	no
Type E5	Legacy-like Promotion	Varies, Support fault suppression	no
Type E5NF	Legacy-like Promotion	Varies, No fault suppression	no
Type E6	Post AVX Promotion	Varies, w/ fault suppression	no
Type E6NF	Post AVX Promotion	Varies, no fault suppression	no
Type E7NM	register-to-register op	none	none
Type E9NF	Miscellaneous 128-bit	Vector-length Specific, no fault suppression	none
Type E10	Non-XF Scalar	Vector Length ignored, w/ fault suppression	none
Type E10NF	Non-XF Scalar	Vector Length ignored, no fault suppression	none
Type E11	VCVTPH2PS	Half Vector Length, w/ fault suppression	yes
Type E11NF	VCVTPS2PH	Half Vector Length, no fault suppression	yes
Type E12	Gather and Scatter Family	VSIB addressing, w/ fault suppression	none
Type E12NP	Gather and Scatter Prefetch Family	VSIB addressing, w/o page fault	none

Table 4-14 lists EVEX-encoded instruction mnemonic by exception classes.

Table 4-14. EVEX Instructions in each Exception Class

Exception Class	Instruction
Type E1	VMOVAPD, VMOVAPS, VMOVDQA32, VMOVDQA64
Type E1NF	VMOVNTDQ, VMOVNTDQA, VMOVNTPD, VMOVNTPS
Type E2	VADDPD, VADDPS, VCMPPD, VCMPPS, VCVTDQ2PS, VCVTPD2DQ, VCVTPD2PS, VCVTPS2DQ, VCVTTPD2DQ, VCVTTPS2DQ, VDIVPD, VDIVPS, VFMADDPD, VFMADDPS, VFMSUBADDPD, VFMSUBADDPS, VFMSUBPD, VFMSUBPS, VFNMADDPD, VFNMADDPS, VFNMSUBPD, VFNMSUBPS, VMAXPD, VMAXPS, VMINPD, VMINPS, VMULPD, VMULPS, VSQRTPD, VSQRTPS, VSUBPD, VSUBPS
	VCVTPD2UDQ, VCVTTPD2DQ, VCVTTPD2UDQ, VCVTTPS2DQ, VCVTTPS2UDQ, VCVTTUDQ2PS, VFIXUPIMMPD, VFIXUPIMMPS, VGETEXPPD, VGETEXPPS, VGETMANTPD, VGETMANTPS, VRNDSCALEPD, VRNDSCALEPS, VSCALEFPD, VSCALEFPS, VRCP28PD, VRCP28PS, VRSQRT28PD, VRSQRT28PS

Table 4-14. EVEX Instructions in each Exception Class (Contd.)

Exception Class	Instruction
Type E3	VADDSD, VADDSS, VCMPSD, VCMPSS, VCVTPS2PD, VCVTSD2SS, VCVTSS2SD, VDIVSD, VDIVSS, VFMADDSD, VFMADDSS, VFMSUBSD, VFMSUBSS, VFNMADDSD, VFNMADDSS, VFNMSUBSD, VFNMSUBSS, VMAXSD, VMAXSS, VMINSD, VMINSS, VMULSD, VMULSS, VSQRTSD, VSQRTSS, VSUBSD, VSUBSS
	, VFIXUPIMMSD, VFIXUPIMMSS, VGETEXPSD, VGETEXPSS, VGETMANTSD, VGETMANTSS, VRNDSCALESD, VRNDSCALESS, VSCALEFSD, VSCALEFSS, VRCP28SD, VRCP28SS, VRSQRT28SD, VRSQRT28SS
Type E3NF	VCOMISD, VCOMISS, VCVTSD2SI, VCVTSI2SD, VCVTSI2SS, VCVTSS2SI, VCVTTSD2SI, VCVTTSS2SI, VUCOMISD, VUCOMISS
	VCVTSD2USI, VCVTTSD2USI, VCVTTSS2USI, VCVTUSI2SD, VCVTUSI2SS
Type E4	VPABSD, VPADDD, VPADDQ, VPANDD, VPANDQ, VPANDND, VPANDNQ, VPCMPEQD, VPCMPEQQ, VPCMPGTD, VPCMPGTQ, VPMAXSD, VPMAXUD, VPMINSD, VPMINUD, VPMULLD, VPMULUDQ, VPMULDQ, VPORD, VPORQ, VPSUBD, VPSUBQ, VPXORD, VPXORQ, VPSLLVD, VPSLLVQ,
	VBLENDMPD, VBLENDMPS, VBLENDMD, VBLENDMD, VBLENDMQ, VPCMPD, VPCMPQ, VPCMPUD, VPCMPUQ, VPLZCNTD, VPLZCNTQ, VPROLD, VPROLQ, (VPSLLD, VPSLLQ, VPSRAD, VPSRLD, VPSRLQ)[1], VPTERNLOGD, VPTERNLOGQ, VPTESTMD, VPTESTMQ, VPTESTNMD, VPTESTNMQ, VRCP14PD, VRCP14PS, VRSQRT14PD, VRSQRT14PS, VPCONFLICTD, VPCONFLICTQ
E4.nb	VMOVUPD, VMOVUPS, VMOVDQU32, VMOVDQU64, VEXPANDPD, VEXPANDPS, VPCOMPRESSD, VPCOMPRESSQ, VPEXPANDD, VPEXPANDQ, , VCOMPRESSPD, VCOMPRESSPS
Type E4NF	VPSHUFD, VPUNPCKHDQ, VPUNPCKHQDQ, VPUNPCKLDQ, VPUNPCKLQDQ, VSHUFPD, VSHUFPS, VUNPCKHPD, VUNPCKHPS, VUNPCKLPD, VUNPCKLPS, VPERMD, VPERMPS, VPERMPD, VPERMQ,
	VALIGND, VALIGNQ, VPERMI2D, VPERMI2PS, VPERMI2PD, VPERMI2Q, VPERMT2D, VPERMT2PS, VPERMT2Q, VPERMT2PD, VPERMILPD, VPERMILPS, VSHUFI32X4, VSHUFF32X4
E4NF.nb	VMOVSHDUP, VMOVSLDUP, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ)[2]
Type E5	VCVTDQ2PD, PMOVSXBD, PMOVSXBQ, PMOVSXWD, PMOVSXWQ, PMOVSXDQ, PMOVZXBD, PMOVZXBQ, PMOVZXWD, PMOVZXWQ, PMOVZXDQ
	VCVTUDQ2PD
Type E5NF	VMOVDDUP
Type E6	VBROADCASTSS, VBROADCASTSD, VBROADCASTF32X4, VBROADCASTI32X4, VPBROADCASTD, VPBROADCASTQ,
	VBROADCASTF32X4, VBROADCASTF64X4,
Type E6NF	VEXTRACTF32X4, VINSERTF32X4, VINSERTF64X4, VINSERTI32X4, VINSERTI64X4, VEXTRACTI32X4, VEXTRACTI64X4, VPBROADCASTMB2Q, VPBROADCASTMW2D, VPMOVQB, VPMOVSQB, VPMOVUSQB, VPMOVQW, VPMOVSQW, VPMOVUSQW, VPMOVQD, VPMOVSQD, VPMOVUSQD, VPMOVDB, VPMOVSDB, VPMOVUSDB, VPMOVDW, VPMOVSDW, VPMOVUSDW
Type E7NM.128[3]	VMOVLHPS, VMOVHLPS
Type E7NM.512[4]	VPBROADCASTD, VPBROADCASTQ,
Type E9NF	VEXTRACTPS, VINSERTPS, VMOVHPD, VMOVHPS, VMOVLPD, VMOVLPS, VMOVD, VMOVQ
Type E10	VMOVSD, VMOVSS, VRCP14SD, VRCP14SS, VRSQRT14SD, VRSQRT14SS,
Type E10NF	(VCVTSI2SD, VCVTUSI2SD)[5]
Type E11	VCVTPH2PS
Type E11NF	VCVTPS2PH
Type E12	VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ, VPSCATTERDD, VPSCATTERDQ, VPSCATTERQD, VPSCATTERQQ, VSCATTERDPD, VSCATTERDPS, VSCATTERQPD, VSCATTERQPS

Table 4-14. EVEX Instructions in each Exception Class (Contd.)

Exception Class	Instruction
Type E12NP	VGATHERPF0DPD, VGATHERPF0DPS, VGATHERPF0QPD, VGATHERPF0QPS, VGATHERPF1DPD, VGATHERPF1DPS, VGATHERPF1QPD, VGATHERPF1QPS, VSCATTERPF0DPD, VSCATTERPF0DPS, VSCATTERPF0QPD, VSCATTERPF0QPS, VSCATTERPF1DPD, VSCATTERPF1DPS, VSCATTERPF1QPD, VSCATTERPF1QPS

NOTES:

1. Operand encoding FV/FVM tupletype with immediate
2. Operand encoding M128 tupletype
3. #UD raised ff EVEX.L'L !=00b (VL=128)
4. #UD raised ff EVEX.L'L !=10b (VL=512)
5. W0 encoding only

4.10.1 Exceptions Type E1 and E1NF of EVEX-Encoded Instructions

EVEX-encoded instructions with memory alignment restrictions, and supporting memory fault suppression follow exception class E1

Table 4-15. Type E1 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0 ▪ If EVEX.L'L != 10b (VL=512)
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'

Table 4-15. Type E1 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X	X	EVEX.512: Memory operand is not 64-byte aligned EVEX.256: Memory operand is not 32-byte aligned EVEX.128: Memory operand is not 16-byte aligned
			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault

EVEX-encoded instructions with memory alignment restrictions, but do not support memory fault suppression follow exception class E1NF

Table 4-16. Type E1NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0 ▪ If EVEX.L'L != 10b (VL=512)
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'

Table 4-16. Type E1NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X	X	EVEX.512: Memory operand is not 64-byte aligned EVEX.256: Memory operand is not 32-byte aligned EVEX.128: Memory operand is not 16-byte aligned
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault

4.10.2 Exceptions Type E2 of EVEX-Encoded Instructions

EVEX-encoded vector instructions with arithmetic semantic follow exception class E2

Table 4-17. Type E2 Class Exception Conditions

Exception	Real	Virtual 8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.L'L != 10b (VL=512)
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form

Table 4-17. Type E2 Class Exception Conditions

Exception	Real	Virtual 8086	Protected and Compatibility	64-bit	Cause of Exception
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1

4.10.3 Exceptions Type E3 and E3NF of EVEX-Encoded Instructions

EVEX-encoded scalar instructions with arithmetic semantic that support memory fault suppression follow exception class E3.

Table 4-18. Type E3 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSX-MMEXCPT[bit 10] = 1

EVEX-encoded scalar instructions with arithmetic semantic that do not support memory fault suppression follow exception class E3NF.

Table 4-19. Type E3NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			EVEX prefix
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1

4.10.4 Exceptions Type E4 and E4NF of EVEX-Encoded Instructions

EVEX-encoded vector instructions that cause no SIMD FP exception and support memory fault suppression follow exception class E4.

Table 4-20. Type E4 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0 and in E4.nb subclass (see E4.nb entries in Table 4-14) ▪ If EVEX.L'L != 10b (VL=512)
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault

EVEX-encoded vector instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E4NF.

Table 4-21. Type E4NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0 and in E4NF.nb subclass (see E4NF.nb entries in Table 4-14) ▪ If EVEX.L'L != 10b (VL=512)
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault

4.10.5 Exceptions Type E5 and E5NF

EVEX-encoded scalar/partial-vector instructions that cause no SIMD FP exception and support memory fault suppression follow exception class E5.

Table 4-22. Type E5 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0 ▪ If EVEX.L'L != 10b (VL=512)
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

EVEX-encoded scalar/partial vector instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E5NF.

Table 4-23. Type E5NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0 ▪ If EVEX.L'L != 10b (VL=512)
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		If an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	If a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

4.10.6 Exceptions Type E6 and E6NF

Table 4-24. Type E6 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0 ▪ If EVEX.L'L != 10b (VL=512)
			X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
			X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	If fault suppression not set, and a page fault
Alignment Check #AC(0)			X	X	For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

EVEX-encoded instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E6NF.

Table 4-25. Type E6NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0 ▪ If EVEX.L'L != 10b (VL=512)
			X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
			X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	For a page fault
Alignment Check #AC(0)			X	X	For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

4.10.7 Exceptions Type E7NM

EVEX-encoded instructions that cause no SIMD FP exception and do not reference memory follow exception class E7NM

Table 4-26. Type E7NM Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0 ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1

4.10.8 Exceptions Type E9 and E9NF

EVEX-encoded vector or partial-vector instructions that do not cause no SIMD FP exception and support memory fault suppression follow exception class E5.

Table 4-27. Type E9 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0 ▪ If EVEX.L'L != 00b (VL=128)
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

EVEX-encoded vector or partial-vector instructions that must be encoded with VEX.L'L = 0, do not cause SIMD FP exception nor support memory fault suppression follow exception class E9NF.

Table 4-28. Type E9NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0 ▪ If EVEX.L'L != 00b (VL=128)
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		If an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	If a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

4.10.9 Exceptions Type E10

EVEX-encoded scalar instructions that ignore EVEX.L'L vector length encoding and do not cause no SIMD FP exception, support memory fault suppression follow exception class E10.

Table 4-29. Type E10 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

EVEX-encoded scalar instructions that must be encoded with VEX.L'L = 0, do not cause SIMD FP exception nor support memory fault suppression follow exception class E5NF.

Table 4-30. Type E10NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

4.10.10 Exception Type E11 and E11NF (VEX-only, mem arg no AC, floating-point exceptions)

EVEX-encoded instructions that can cause SIMD FP exception, memory operand support fault suppression but do not cause #AC follow exception class E11

Table 4-31. Type E11 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0 ▪ If EVEX.L'L != 10b (VL=512)
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF (fault-code)		X	X	X	If fault suppression not set, and a page fault
SIMD Floating-Point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} not set, and CR4.OSXMMEX-CPT[bit 10] = 1

EVEX-encoded instructions that can cause SIMD FP exception, memory operand do not support fault suppression and do not cause #AC follow exception class E11NF.

Table 4-32. Type E11NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0 ▪ If EVEX.L'L != 10b (VL=512)
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF (fault-code)		X	X	X	For a page fault
SIMD Floating-Point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} not set, and CR4.OSXMMEXCPT[bit 10] = 1

4.10.11 Exception Type E12 (VSIB mem arg, no AC, no floating-point exceptions)

Table 4-33. Type E12 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0 ▪ If EVEX.L'L != 10b (VL=512)
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
	X	X	X	NA	If address size attribute is 16 bit
	X	X	X	X	If ModR/M.mod = '11b'
	X	X	X	X	If ModR/M.rm != '100b'
	X	X	X	X	If any corresponding CPUID feature flag is '0'
	X	X	X	X	If any vector register is used more than once between the destination register, mask register and the index register in VSIB addressing.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF (fault-code)		X	X	X	For a page fault

EVEX-encoded prefetch instructions that do not cause #PF follow exception class E12NP.

Table 4-34. Type E12NP Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			if EVEX prefix present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10, ▪ Opmask encoding #UD condition of Table 4-11, ▪ If EVEX.b != 0 ▪ If EVEX.L'L != 10b (VL=512)
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
	X	X	X	NA	If address size attribute is 16 bit
	X	X	X	X	If ModR/M.mod = '11b'
	X	X	X	X	If ModR/M.rm != '100b'
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH

4.11 EXCEPTION CLASSIFICATIONS OF OPMASK INSTRUCTIONS

The exception behavior of VEX-encoded opmask instructions are listed below.

Exception conditions of Opmask instructions that do not address memory are listed as Type K20.

Table 4-35. TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If relevant CPUID feature flag is '0'
	X	X			If a VEX prefix is present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10,
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
			X	X	If ModRM:[7:6] != 11b

Exception conditions of Opmask instructions that address memory are listed as Type K21.

Table 4-36. TYPE K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory)

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If relevant CPUID feature flag is '0'
	X	X			If a VEX prefix is present
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 4-8 not met, ▪ Opcode independent #UD condition in Table 4-9, ▪ Operand encoding #UD conditions in Table 4-10,
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
Stack, SS(0)	X	X	X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

This page was intentionally left blank.

CHAPTER 5
INSTRUCTION SET REFERENCE, A-Z

Instructions described in this document follow the general documentation convention established in *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A* and *2B*. Additional notations and conventions adopted in this document are listed in *Section 5.1. Section 5.1.5.1* covers supplemental information that applies to a specific subset of instructions.

5.1 INTERPRETING INSTRUCTION REFERENCE PAGES

This section describes the format of information contained in the instruction reference pages in this chapter. It explains notational conventions and abbreviations used in these sections that are outside of those conventions described in *Section 3.1* of *the Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*.

5.1.1 Instruction Format

The following is an example of the format used for each instruction description in this chapter. The table below provides an example summary table:

ADDPS—Add Packed Single-Precision Floating-Point Values (THIS IS AN EXAMPLE)

Opcode/ Instruction	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 58 /r ADDPS xmm1, xmm2/m128	V/V	SSE	Add packed single-precision floating-point values from xmm2/mem to xmm1 and store result in xmm1.
VEX.NDS.128.0F 58 /r VADDPS xmm1,xmm2, xmm3/m128	V/V	AVX	Add packed single-precision floating-point values from xmm3/mem to xmm2 and store result in xmm1.
VEX.NDS.256.0F 58 /r VADDPS ymm1, ymm2, ymm3/m256	V/V	AVX	Add packed single-precision floating-point values from ymm3/mem to ymm2 and store result in ymm1.
VEX.L1.0F.W0 41 /r KANDW k1, k2, k3	V/V	AVX512F	Bitwise AND word masks k2 and k3 and place result in k1.
EVEX.NDS.512.0F.W0 58 /r VADDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	V/V	AVX512F	Add packed single-precision floating-point values from zmm3/m512/m32bcst with zmm2 and store result in zmm1 with writemask k1.

5.1.2 Opcode Column in the Instruction Summary Table

For notation and conventions applicable to instructions that do not use VEX prefix, consult *Section 3.1* of the *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*.

In the Instruction Summary Table, the Opcode column presents each instruction encoded using the VEX prefix in following form (including the modR/M byte if applicable, the immediate byte if applicable):

VEX.[NDS/NDD/DS].[128,256,L0,L1,LIG].[66,F2,F3].0F/0F3A/0F38.[W0,W1,WIG] opcode [/r] [/ib,/is4]

- **VEX:** indicates the presence of the VEX prefix is required. The VEX prefix can be encoded using the three-byte form (the first byte is C4H), or using the two-byte form (the first byte is C5H). The two-byte form of VEX only

applies to those instructions that do not require the following fields to be encoded: VEX.mmmmm, VEX.W, VEX.X, VEX.B.

The encoding of various sub-fields of the VEX prefix is described using the following notations:

— **NDS, NDD, DDS:** implies that VEX.vvvv field is valid for the encoding of an operand. It may specify either the source register (NDS) or the destination register (NDD). If both NDS and NDD are absent (i.e. VEX.vvvv does not encode an operand), VEX.vvvv must be 1111b. The VEX.vvvv field can be encoded using either the 2-byte or 3-byte form of the VEX prefix. DDS expresses a syntax where vvvv encodes the second source register in a three-operand instruction syntax where the content of first source register will be overwritten by the result.

— **128,256,L0,L1:** VEX.L fields can be 0 (denoted by VEX.128 or VEX.L0 for mask instructions) or 1 (denoted by VEX.256 or VEX.L1 for mask instructions). The VEX.L field can be encoded using either the 2-byte or 3-byte form of the VEX prefix. The presence of the notation VEX.256 or VEX.128 in the opcode column should be interpreted as follows:

- If VEX.256 is present in the opcode column: The semantics of the instruction must be encoded with VEX.L = 1. An attempt to encode this instruction with VEX.L= 0 can result in one of two situations: (a) if VEX.128 version is defined, the processor will behave according to the defined VEX.128 behavior; (b) an #UD occurs if there is no VEX.128 version defined.
- If VEX.128 is present in the opcode column but there is no VEX.256 version defined for the same opcode byte: Three situations apply: (a) For VEX-encoded, 128-bit SIMD integer instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception; (b) For VEX-encoded, 128-bit packed floating-point instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception (e.g. VMOVLPS); (c) For VEX-encoded, scalar, SIMD floating-point instructions, software should encode the instruction with VEX.L = 0 to ensure software compatibility with future processor generations. Scalar SIMD floating-point instruction can be distinguished from the mnemonic of the instruction. Generally, the last two letters of the instruction mnemonic would be either "SS", "SD", or "SI" for SIMD floating-point conversion instructions, except VBROADCASTSx are unique cases.
- VEX.L0 and VEX.L1 notations are used in the case of masking instructions such as KANDW since the VEX.L bit is not used to distinguish between the 128-bit and 256-bit forms for these instructions. Instead, this bit is used to distinguish between the two operand form (VEX.L0) and the three operand form (VEX.L1) of the same mask instruction.
- If VEX.L0 is present in the opcode column: The semantics of the instruction must be encoded with VEX.L = 0. An attempt to encode this instruction with VEX.L= 1 can result in one of two situations: (a) if VEX.L1 version is defined, the processor will behave according to the defined VEX.L1 behavior; (b) an #UD occurs if there is no VEX.L1 version defined.
- If VEX.L1 is present in the opcode column: The semantics of the instruction must be encoded with VEX.L = 1. An attempt to encode this instruction with VEX.L= 0 can result in one of two situations: (a) if VEX.L0 version is defined, the processor will behave according to the defined VEX.L1 behavior; (b) an #UD occurs if there is no VEX.L0 version defined.
- **LIG:** VEX.L bit ignored

— **66,F2,F3:** The presence or absence of these value maps to the VEX.pp field encodings. If absent, this corresponds to VEX.pp=00B. If present, the corresponding VEX.pp value affects the "opcode" byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix. The VEX.pp field may be encoded using either the 2-byte or 3-byte form of the VEX prefix.

— **0F,0F3A,0F38:** The presence maps to a valid encoding of the VEX.mmmmm field. Only three encoded values of VEX.mmmmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH and 0F38H. The effect of a valid VEX.mmmmm encoding on the ensuing opcode byte is same as if the corresponding escape byte sequence on the ensuing opcode byte for non-VEX encoded instructions. Thus a valid encoding of VEX.mmmmm may be consider as an implies escape byte sequence of either 0FH, 0F3AH or 0F38H. The VEX.mmmmm field must be encoded using the 3-byte form of VEX prefix.

  - — **0F,0F3A,0F38 and 2-byte/3-byte VEX.** The presence of 0F3A and 0F38 in the opcode column implies that opcode can only be encoded by the three-byte form of VEX. The presence of 0F in the opcode column does not preclude the opcode to be encoded by the two-byte of VEX if the semantics of the opcode does not require any subfield of VEX not present in the two-byte form of the VEX prefix.
  - — **W0:** VEX.W=0.
  - — **W1:** VEX.W=1.
  - — **WIG:** VEX.W bit ignored
  - — The presence of W0/W1 in the opcode column applies to two situations: (a) it is treated as an extended opcode bit, (b) the instruction semantics support an operand size promotion to 64-bit of a general-purpose register operand or a 32-bit memory operand. The presence of W1 in the opcode column implies the opcode must be encoded using the 3-byte form of the VEX prefix. The presence of W0 in the opcode column does not preclude the opcode to be encoded using the C5H form of the VEX prefix, if the semantics of the opcode does not require other VEX subfields not present in the two-byte form of the VEX prefix. If neither W0 or W1 is present, the instruction may be encoded using either the two-byte form (if the opcode semantic does not require VEX subfields not present in the two-byte form of VEX) or the three-byte form of VEX. Encoding an instruction using the two-byte form of VEX is equivalent to W0.
- **opcode:** Instruction opcode.
- **/ib:** An 8-bit immediate byte is present and used as one of the instructions operands.
- **/is4:** An 8-bit immediate byte is present containing a source register specifier in imm[7:4] and instruction-specific payload in imm[3:0].
- **imz2**: Part of the is4 immediate byte providing control functions that apply to two-source permute instructions

In general, the encoding o f VEX.R, VEX.X, VEX.B field are not shown explicitly in the opcode column.

EVEX.[NDS].[128,256,512,LIG].[66,F2,F3].0F/0F3A/0F38.[W0,W1,WIG] opcode [/r] [/ib,/is4]

- **EVEX:** The EVEX prefix is encoded using the four-byte form (the first byte is 62H). Refer to Section 4.2 for more detail on the EVEX prefix.

  The encoding of various sub-fields of the EVEX prefix is described using the following notations:

  - — **NDS, NDD:** implies that EVEX.vvvv (and EVEX.v') field is valid for the encoding of an operand. It may specify either the source register (NDS) or the destination register (NDD). If both NDS and NDD absent (i.e. EVEX.vvvv does not encode an operand), EVEX.vvvv must be 1111b (and EVEX.v' must be 1b).
  - — **128, 256, 512, LIG:** This corresponds to the vector length; three values are allowed by EVEX: 512-bit, 256-bit and 128-bit. Alternatively, vector length is ignored (LIG) for certain instructions; this is the in particular for scalar instructions.
  - — **66,F2,F3:** The presence of these value maps to the EVEX.pp field encodings. The corresponding VEX.pp value affects the "opcode" byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix.
  - — **0F,0F3A,0F38:** The presence maps to a valid encoding of the EVEX.mmm field. Only three encoded values of EVEX.mmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH and 0F38H. The effect of a valid EVEX.mmm encoding on the ensuing opcode byte is same as if the corresponding escape byte sequence on the ensuing opcode byte for non-EVEX encoded instructions. Thus a valid encoding of EVEX.mmm may be consider as an implies escape byte sequence of either 0FH, 0F3AH or 0F38H.
  - — **W0:** EVEX.W=0.
  - — **W1:** EVEX.W=1.
  - — **WIG:** EVEX.W bit ignored
- **opcode:** Instruction opcode.
- **/is4:** An 8-bit immediate byte is present containing a source register specifier in imm[7:4] and instruction-specific payload in imm[3:0].
- **imz2**: Part of the is4 immediate byte providing control functions that apply to two-source permute instructions

- In general, the encoding o f EVEX.R and R', EVEX.X and X', and EVEX.B and B' fields are not shown explicitly in the opcode column.

5.1.3 Instruction Column in the Instruction Summary Table

<additions to the SDM section with the same title>

- ymm — a YMM register. The 256-bit YMM registers are: YMM0 through YMM7; YMM8 through YMM15 are available in 64-bit mode. YMM16 through YMM31 are available in 64-bit mode via EVEX prefix.
- m256 — A 32-byte operand in memory. This nomenclature is used only with AVX and FMA instructions.
- **ymm/m256** - a YMM register or 256-bit memory operand.
- **<YMM0>:** indicates use of the YMM0 register as an implicit argument.
- zmm — a ZMM register. The 512-bit ZMM registers are: ZMM0 through ZMM7; ZMM8 through ZMM15 are available in 64-bit mode. ZMM16 through ZMM31 are available in 64-bit mode via EVEX prefix.
- m512 — A 64-byte operand in memory. This nomenclature is used only with AVX and FMA instructions.
- **zmm/m512** — a ZMM register or 512-bit memory operand.
- **{k1}{z}** — a mask register used as instruction writemask. The 64-bit k registers are: k1 through k7. Writemask specification is available exclusively via EVEX prefix. The masking can either be done as a merging-masking, where the old values are preserved for masked out elements or as a zeroing masking. The type of masking is determined by using the EVEX.z bit.
- **{k1}** — without {z}: a mask register used as instruction writemask for instructions that do not allow zeroing-masking but support merging-masking. This corresponds to instructions that require the value of the aaa field to be different than 0 (e.g., gather) and store-type instructions which allow only merging-masking.
- **k1** — a mask register used as a regular operand (either destination our source). The 64-bit k registers are: k0 through k7.
- mV — a vector memory operand; the operand size is dependent on the instruction.
- vm32{x,yz} — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 32-bit index value in an XMM register (vm32x), a YMM register (vm32y) or a ZMM register (vm32z).
- vm64{x,yz} — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 64-bit index value in an XMM register (vm64x), a YMM register (vm64y) or a ZMM register (vm64z).
- **zmm/m512/m32bcst** — an operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 32-bit memory location.
- **zmm/m512/m64bcst** — an operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 64-bit memory location.
- **<ZMM0>:** indicates use of the ZMM0 register as an implicit argument.
- **{er}** indicates support for embedded rounding control, which is only applicable to the register-register form of the instruction. This also implies support for SAE (Suppress All Exceptions).
- **{sae}** indicates support for SAE (Suppress All Exceptions). This is used for instructions that support SAE, but do not support embedded rounding control.
- SRC1 - Denotes the first source operand in the instruction syntax of an instruction encoded with the EVEX prefix and having two or more source operands.
- SRC2 - Denotes the second source operand in the instruction syntax of an instruction encoded with the EVEX prefix and having two or more source operands.
- **SRC3 -** Denotes the third source operand in the instruction syntax of an instruction encoded with the EVEX prefix and having three source operands.
- **SRC -** The source in an single-source instruction.
- **DST -** the destination in an instruction. This field is encoded by reg_field.

5.1.4 64/32 bit Mode Support column in the Instruction Summary Table

The "64/32 bit Mode Support" column in the Instruction Summary table indicates whether an opcode sequence is supported in 64-bit or the Compatibility/other IA32 modes.

The 64-bit mode support is to the left of the 'slash' and has the following notation:

- **V** — Supported.
- **I** — Not supported.
- **N.E**. — Indicates an instruction syntax is not encodable in 64-bit mode (it may represent part of a sequence of valid instructions in other modes).
- **N.P.** — Indicates the REX prefix does not affect the legacy instruction in 64-bit mode.
- **N.I.** — Indicates the opcode is treated as a new instruction in 64-bit mode.
- **N.S.** — Indicates an instruction syntax that requires an address override prefix in 64-bit mode and is not supported. Using an address override prefix in 64-bit mode may result in model-specific execution behavior.

The compatibility/Legacy mode support is to the right of the 'slash' and has the following notation:

- **V —** Supported.
- **I —** Not supported.
- **N.E. —** Indicates an Intel 64 instruction mnemonics/syntax that is not encodable; the opcode sequence is not applicable as an individual instruction in compatibility mode or IA-32 mode. The opcode may represent a valid sequence of legacy IA-32 instructions.

5.1.5 CPUID Support column in the Instruction Summary Table

The fourth column holds abbreviated CPUID feature flags (e.g. appropriate bit in CPUID.1.ECX, CPUID.1.EDX for SSE/SSE2/SSE3/SSSE3/SSE4.1/SSE4.2/AVX support) that indicate processor support for the instruction. If the corresponding flag is '0', the instruction will #UD.

5.1.5.1 Operand Encoding Column in the Instruction Summary Table

The "operand encoding" column is abbreviated as Op/En in the Instruction Summary table heading. Instruction operand encoding information is provided for each assembly instruction syntax using a letter to cross reference to a row entry in the operand encoding definition table that follows the instruction summary table. The operand encoding table in each instruction reference page lists each instruction operand (according to each instruction syntax and operand ordering shown in the instruction column) relative to the ModRM byte, VEX.vvvv field or additional operand encoding placement.

EVEX encoded instructions employ compressed disp8*N, where N is defined in Table 4-5 and Table 4-6, according to tupletypes. The Op/En column of an EVEX encoded instruction uses the corresponding tupletype abbreviation.

NOTES

The letters in the Op/En column of an instruction apply ONLY to the encoding definition table immediately following the instruction summary table.

In the encoding definition table, the letter 'r' within a pair of parenthesis denotes the content of the operand will be read by the processor. The letter 'w' within a pair of parenthesis denotes the content of the operand will be updated by the processor.

5.2 SUMMARY OF TERMS

- **"Legacy SSE":** Refers to SSE, SSE2, SSE3, SSSE3, SSE4, and any future instruction sets referencing XMM registers and encoded without a VEX prefix.

- **XGETBV, XSETBV, XSAVE, XRSTOR** are defined in *Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A* and *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.
- VEX: refers to a two-byte or three-byte prefix. AVX and FMA instructions are encoded using a VEX prefix.
- **EVEX:** refers to a four-byte prefix. AVX512F instructions are encoded using an EVEX prefix.
- **VEX.vvvv**. The VEX bit field specifying a source or destination register (in 1's complement form).
- **rm_field**: shorthand for the ModR/M *r/m* field and any REX.B
- reg_field: shorthand for the ModR/M *reg* field and any REX.R

5.3 INSTRUCTION SET REFERENCE

<Only instructions modified by AVX512F are included.>

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	RM	V/V	SSE2	Add packed double-precision floating-point values from xmm2/mem to xmm1 and store result in xmm1.
VEX.NDS.128.66.0F.WIG 58 /r VADDPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Add packed double-precision floating-point values from xmm3/mem to xmm2 and store result in xmm1.
VEX.NDS.256.66.0F.WIG 58 /r VADDPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Add packed double-precision floating-point values from ymm3/mem to ymm2 and store result in ymm1.
EVEX.NDS.512.66.0F.W1 58 /r VADDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Add packed double-precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Add 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 results in the destination operand.

EVEX encoded versions: The first source operand is a ZMM register. The second 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 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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VADDPD (EVEX encoded versions) when src2 operand is a vector register

```
(KL, VL) = (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
```

```
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← 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
```

VADDPD (EVEX encoded versions) when src2 operand is a memory source
(KL, VL) = (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
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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 __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 Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

ADDPS—Add Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 58 /r ADDPS xmm1, xmm2/m128	RM	V/V	SSE	Add packed single-precision floating-point values from xmm2/m128 to xmm1 and store result in xmm1.
VEX.NDS.128.0F.WIG 58 /r VADDPS xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Add packed single-precision floating-point values from xmm3/m128 to xmm2 and store result in xmm1.
VEX.NDS.256.0F.WIG 58 /r VADDPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Add packed single-precision floating-point values from ymm3/m256 to ymm2 and store result in ymm1.
EVEX.NDS.512.0F.W0 58 /r VADDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	FV	V/V	AVX512F	Add packed single-precision floating-point values from zmm3/m512/m32bcst to zmm2 and store result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Add 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 results in the destination operand.

EVEX encoded versions: The first source operand is a ZMM register. 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 destination operand is a ZMM 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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

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

```
(KL, VL) = (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
```

```
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← 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;
```

VADDPS (EVEX encoded versions) when src2 operand is a memory source
(KL, VL) = (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;
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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 __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 Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

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	RM	V/V	SSE2	Add the low double-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.NDS.128.F2.0F.WIG 58 /r VADDSD xmm1, xmm2, xmm3/m64	RVM	V/V	AVX	Add the low double-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F2.0F.W1 58 /r VADDSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Add the low double-precision floating-point value from xmm3/m64 to xmm2 and store the result in xmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S-RVM	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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[MAX_VL-1:128] ← 0

VADDSD (VEX.128 encoded version)
DEST[63:0] ←SRC1[63:0] + SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAX_VL-1:128] ←0

ADDSD (128-bit Legacy SSE version)
DEST[63:0] ←DEST[63:0] + SRC[63:0]
DEST[MAX_VL-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 Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

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	RM	V/V	SSE	Add the low single-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.NDS.128.F3.0F.WIG 58 /r VADDSS xmm1,xmm2, xmm3/m32	RVM	V/V	AVX	Add the low single-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F3.0F.W0 58 /r VADDSS xmm1{k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Add the low single-precision floating-point value from xmm3/m32 to xmm2 and store the result in xmm1with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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[MAX_VL-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[MAX_VL-1:128] ←0

ADDSS DEST, SRC (128-bit Legacy SSE version)
DEST[31:0] ←DEST[31:0] + SRC[31:0]
DEST[MAX_VL-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 Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors

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

Instruction Operand Encoding

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

Description

Concatenates and shifts right doubleword/quadword elements of the first source operand (the second operand) and the second source operand (the third operand). The result of the low 512-bit vector is written to the destination operand (the first operand) using the writemask k1. 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 32/64-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) = (16, 512)                    ;for 512-bit forms

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;
```

VALIGNQ (EVEX encoded versions)

```
(KL, VL) = (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
SHIFT = imm8[2:0]
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;
```

Intel C/C++ Compiler Intrinsic Equivalent

VALIGND __m512i _mm512_alignr_epi32(__m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_mask_alignr_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_maskz_alignr_epi32(__mmask16 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_alignr_epi64(__m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_mask_alignr_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_maskz_alignr_epi64(__mmask8 k, __m512i a, __m512i b, int cnt);

Exceptions

See Exceptions Type E4NF.

VBLENDMPD—Blend Float64 Vectors Using an OpMask Control

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F38.W1 65 /r VBLENDMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Blend double-precision vector zmm2 and double-precision vector zmm3/m512/m64bcst using control mask k1, and store the result in zmm1.

Instruction Operand Encoding

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

Description

Performs an element-by-element blending between float64 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 registers. The second 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 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) = (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
```

Intel C/C++ Compiler Intrinsic Equivalent

VBLENDMPD __m512d _mm512_mask_blend_pd(__mmask8 k, __m512d a, __m512d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VBLENDMPS—Blend Float32 Vectors Using an OpMask Control

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F38.W0 65 /r VBLENDMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Blend single-precision vector zmm2 and single-precision vector zmm3/m512/m32bcst using k1 as select control and store the result in zmm1.

Instruction Operand Encoding

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

Description

Performs an element-by-element blending between float32 elements in the first source operand (the second operand) with the elements from 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

VBLENDMPS (EVEX encoded versions)

```
(KL, VL) = (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
```

Intel C/C++ Compiler Intrinsic Equivalent

VBLENDMPS __m512 _mm512_mask_blend_ps(__mmask16 k, __m512 a, __m512 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VPBLENDMD—Blend Int32 Vectors Using an OpMask Control

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F38.W0 64 /r VPBLENDMD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Blend doubleword integer vector zmm2 and doubleword vector zmm3/m512/m32bcst using control mask k1, and store the result in zmm1.

Instruction Operand Encoding

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

Description

Performs an element-by-element blending of doubleword 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) = (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
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDMD __m512i _mm512_mask_blend_epi32(__mmask16 k, __m512i a, __m512i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VPBLENDMQ—Blend Int64 Vectors Using an OpMask Control

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F38.W1 64 /r VPBLENDMQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Blend quadword integer vector zmm2 and quadword vector zmm3/m512/m64bcst using control mask k1, and store the result in zmm1.

Instruction Operand Encoding

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

Description

Performs an element-by-element blending of quadword elements between the first source operand (the second operand) and the elements of the second source operand (the third operand) using an opmask as select control. The blended result is written into the destination register.

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 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 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

VPBLENDMQ (EVEX encoded versions)

```
(KL, VL) = (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
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDMQ __m512i _mm512_mask_blend_epi64(__mmask8 k, __m512i a, __m512i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VBROADCAST—Load with Broadcast Floating-Point Data

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1, m32	RM	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	RM	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	RM	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	RM	V/V	AVX	Broadcast 128 bits of floating-point data in mem to low and high 128-bits in ymm1.
EVEX.512.66.0F38.W1 19 /r VBROADCASTSD zmm1 {k1}{z}, xmm2/m64	T1S	V/V	AVX512F	Broadcast low double-precision floating-point element in xmm2/m64 to eight locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 18 /r VBROADCASTSS zmm1 {k1}{z}, xmm2/m32	T1S	V/V	AVX512F	Broadcast low single-precision floating-point element in xmm2/m32 to all locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 1A /r VBROADCASTF32X4 zmm1 {k1}{z}, mV	T4	V/V	AVX512F	Broadcast 128 bits of 4 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}, mV	T4	V/V	AVX512F	Broadcast 256 bits of 4 double-precision floating-point data in mem to locations in zmm1using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S,T4	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

EVEX-encoded versions: The destination operand is a ZMM 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.

VBROADCASTF32X4/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 ZMM register updated according to the writemask k1. The source operand is either a register or 128-bit/256-bit memory location.

VBROADCASTF32X4 have 32-bit granularity. VBROADCASTF64X4 have 64-bit granularity.

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

If VBROADCASTSD or VBROADCASTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

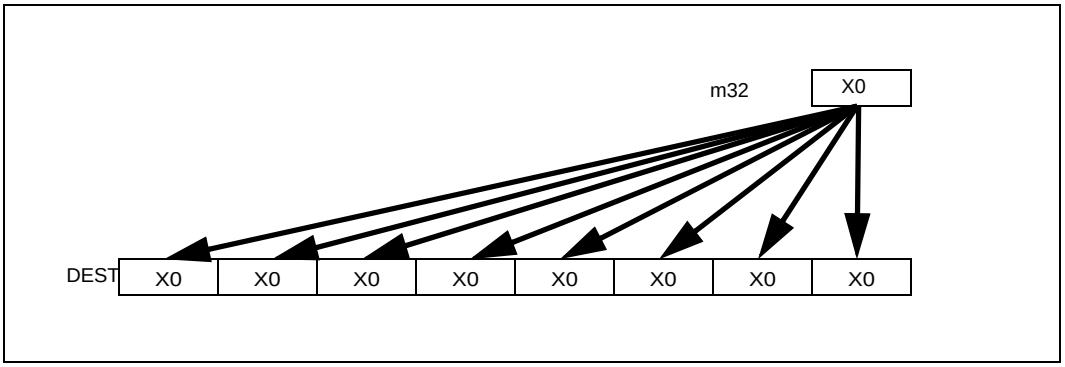


Figure 5-1. VBROADCASTSS Operation (VEX.256 encoded version)

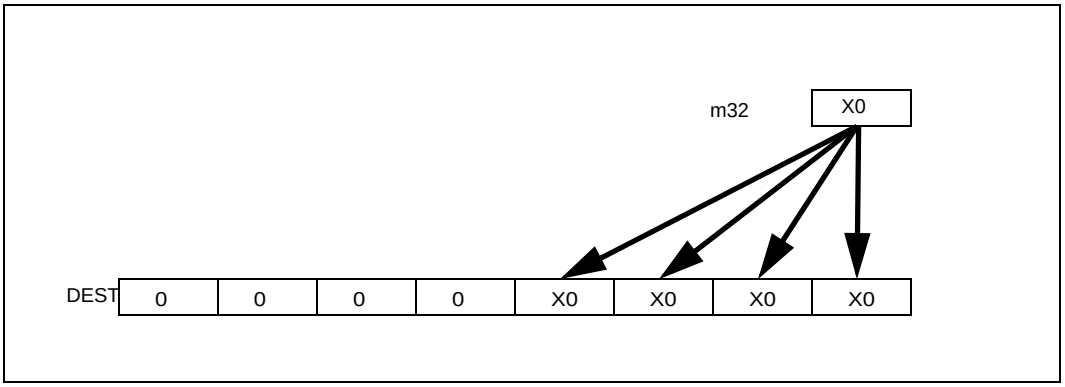


Figure 5-2. VBROADCASTSS Operation (128-bit version)

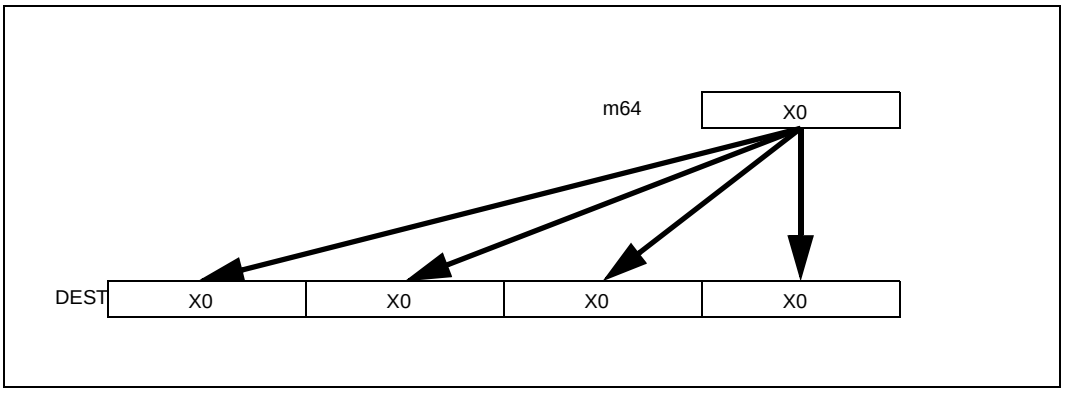


Figure 5-3. VBROADCASTSD Operation (256-bit version)

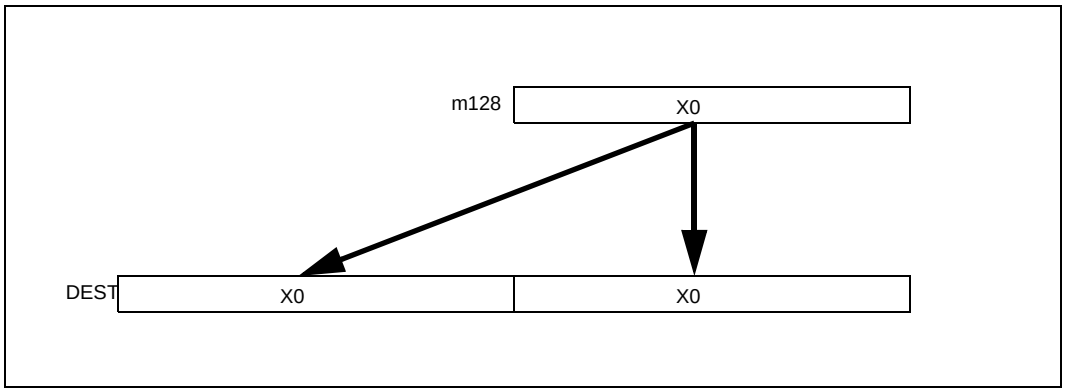


Figure 5-4. VBROADCASTF128 Operation (256-bit version)

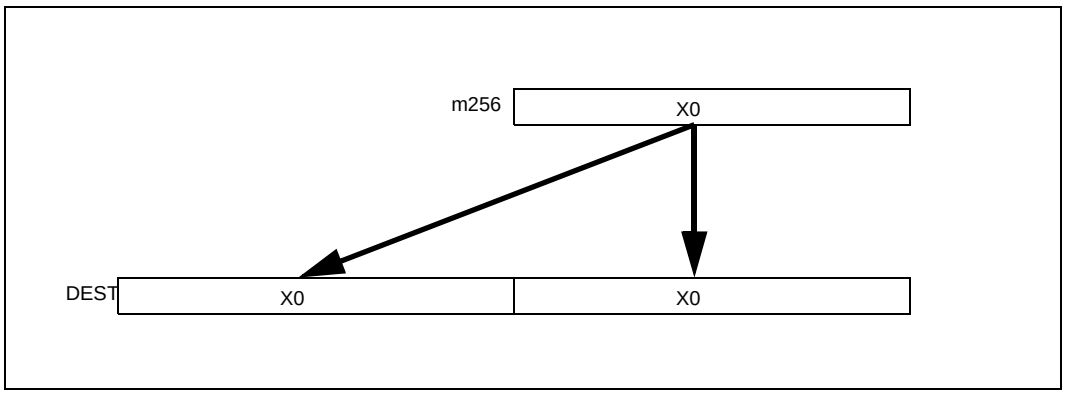


Figure 5-5. VBROADCASTF64X4 Operation (512-bit version)

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[MAX_VL-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[MAX_VL-1:256] ← 0
```

VBROADCASTSS (EVEX encoded versions)

```
(KL, VL) = (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
```

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[MAX_VL-1:256] ← 0
```

VBROADCASTSD (EVEX encoded versions)

```
(KL, VL) = (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
```

VBROADCASTF128 (VEX.256 encoded version)

```
temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[255:128] ← temp
DEST[MAX_VL-1:256] ← 0
```

VBROADCASTF32X4 (EVEX encoded versions)

```
(KL, VL) = (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
```

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
```

Intel C/C++ Compiler Intrinsic Equivalent

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);
VBROADCASTF64x4 __m512 _mm512_broadcast_f64x4(__m256d a);
VBROADCASTF64x4 __m512 _mm512_mask_broadcast_f64x4(__m512d s, __mmask8 k, __m256d a);
VBROADCASTF64x4 __m512 _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);
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 __m128 _mm_broadcast_ss(float *a);
VBROADCASTSS __m256 _mm256_broadcast_ss(float *a);
VBROADCASTSD __m256d _mm256_broadcast_sd(double *a);
VBROADCASTF128 __m256 _mm256_broadcast_ps(__m128 * a);
VBROADCASTF128 __m256d _mm256_broadcast_pd(__m128d * a);

Exceptions

VEX-encoded instructions, see Exceptions Type 6; additionally

#UD	If VEX.L = 0 for VBROADCASTSD.
	If VEX.L = 0 for VBROADCASTF128.

EVEX-encoded instructions, see Exceptions Type E6.

VPBROADCASTD/VPBROADCASTQ—Load with Broadcast Integer Data from GPR

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 7C /r VPBROADCASTD zmm1 {k1}{z}, r32	T1S	V/V	AVX512F	Broadcast a 32-bit value from a GPR to all double-words in the 512-bit destination subject to writemask k1.
EVEX.512.66.0F38.W1 7C /r VPBROADCASTQ zmm1 {k1}{z}, r64	T1S	V/N.E.[1]	AVX512F	Broadcast a 64-bit value from a GPR to all quad-words in the 512-bit destination subject to writemask k1.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Broadcasts a 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

VPBROADCASTD (EVEX encoded versions)

```
(KL, VL) = (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
```

VPBROADCASTQ (EVEX encoded versions)

```
(KL, VL) = (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
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTD __m512i _mm512_mask_set1_epi32(__m512i s, __mmask16 k, int a);
VPBROADCASTD __m512i _mm512_maskz_set1_epi32(__mmask16 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);

Exceptions

EVEX-encoded instructions, see Exceptions Type E7NM.

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	RM	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	RM	V/V	AVX2	Broadcast a byte integer in the source operand to thirty-two locations in ymm1.
VEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1, xmm2/m16	RM	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	RM	V/V	AVX2	Broadcast a word integer in the source operand to sixteen locations in ymm1.
VEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1, xmm2/m32	RM	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	RM	V/V	AVX2	Broadcast a dword integer in the source operand to eight locations in ymm1.
EVEX.512.66.0F38.W0 58 /r VPBROADCASTD zmm1 {k1}{z}, xmm2/m32	T1S	V/V	AVX512F	Broadcast a dword integer in the source operand to locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 59 /r VPBROADCASTQ xmm1, xmm2/m64	RM	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	RM	V/V	AVX2	Broadcast a qword element in source operand to four locations in ymm1.
EVEX.512.66.0F38.W1 59 /r VPBROADCASTQ zmm1 {k1}{z}, xmm2/m64	T1S	V/V	AVX512F	Broadcast a qword element in source operand to locations in zmm1 subject to writemask k1.
VEX.256.66.0F38.W0 5A /r VBROADCASTI128 ymm1, m128	RM	V/V	AVX2	Broadcast 128 bits of integer data in mem to low and high 128-bits in ymm1.
EVEX.512.66.0F38.W0 5A /r VBROADCASTI32X4 zmm1 {k1}{z}, m128	T4	V/V	AVX512F	Broadcast 128 bits of 4 doubleword integer data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W1 5B /r VBROADCASTI64X4 zmm1 {k1}{z}, m256	T4	V/V	AVX512F	Broadcast 256 bits of 4 quadword integer data in mem to locations in zmm1using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S, T4	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

VEX256-encoded VPBROADCASTB/W/D/Q: The source operand is 8-bit, 16-bit, 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. The destination operand is a YMM register. VPBROADCASTI128 support the source operand of 128-bit memory location. Register source encodings for VPBROADCASTI128 is reserved and will #UD. Bits (MAX_VL-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 register and updated according to the writemask k1.

VPBROADCASTI32X4 and VPBROADCASTI64X4: The destination operand is a ZMM register and updated according to the writemask k1. The source operand is 128-bit or 256-bit memory location. Register source encodings for VBROADCASTI32X4 and VBROADCASTI64X4 are reserved and will #UD.

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

If VPBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

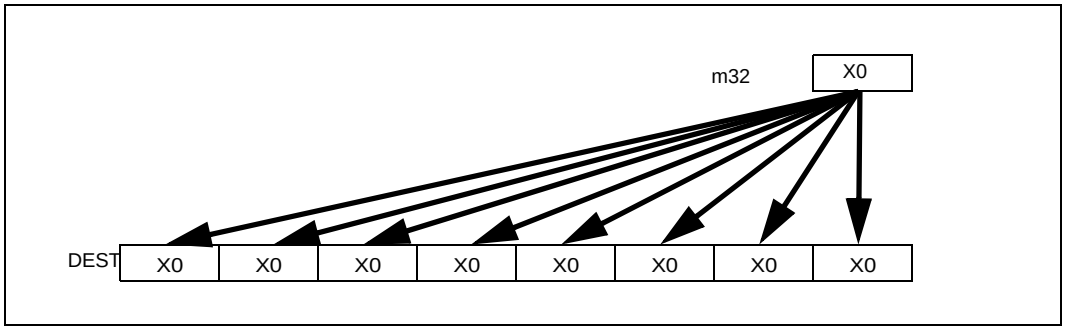


Figure 5-6. VPBROADCASTD Operation (VEX.256 encoded version)

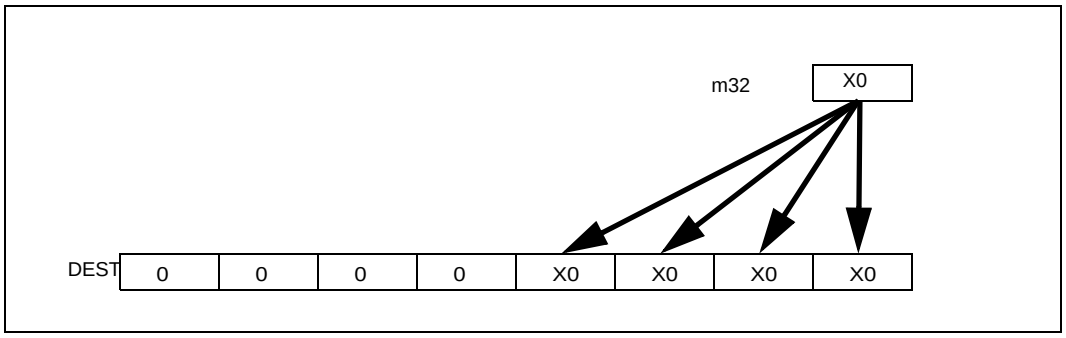


Figure 5-7. VPBROADCASTD Operation (128-bit version)

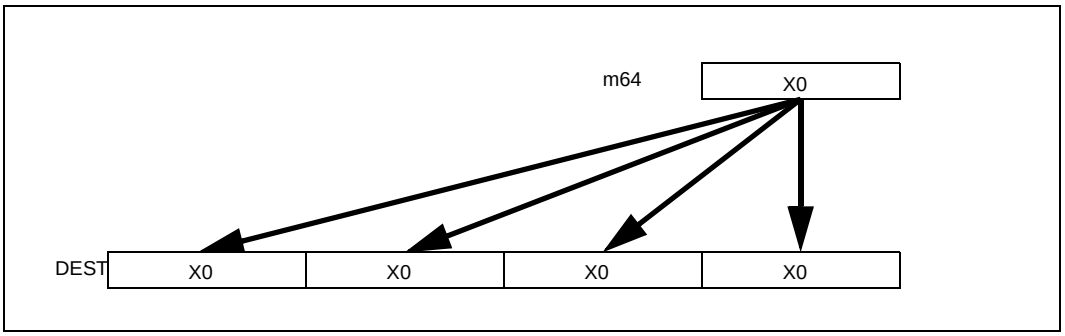


Figure 5-8. VPBROADCASTQ Operation (256-bit version)

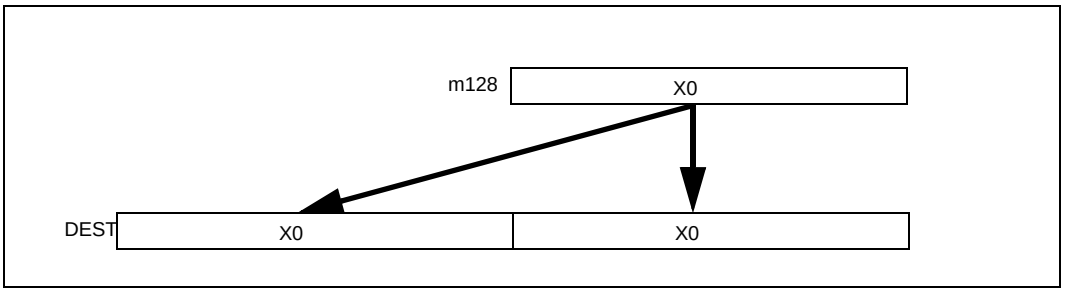


Figure 5-9. VBROADCASTI128 Operation (256-bit version)

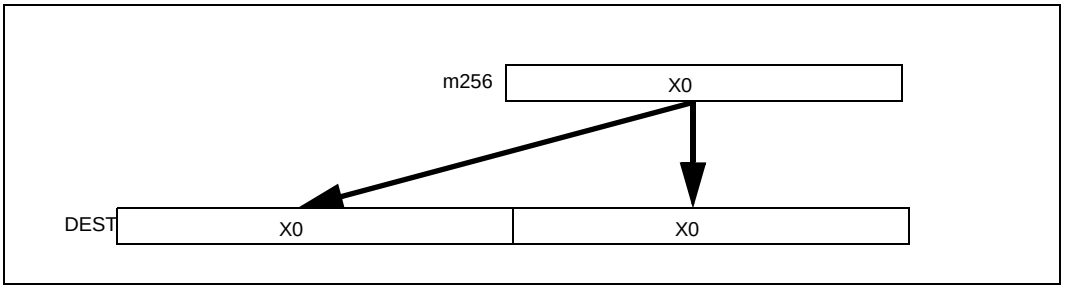


Figure 5-10. VBROADCASTI256 Operation (512-bit version)

Operation

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[MAX_VL-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[MAX_VL-1:256] ← 0
```

VPBROADCASTD (EVEX encoded versions)

```
(KL, VL) = (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
```

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[MAX_VL-1:256] ← 0
```

VPBROADCASTQ (EVEX encoded versions)

```
(KL, VL) = (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
```

VBROADCASTI128 (VEX.256 encoded version)

```
temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[255:128] ← temp
DEST[MAX_VL-1:256] ← 0
```

VBROADCASTI32X4 (EVEX encoded versions)

```
(KL, VL) =(16, 512)
```

```
FOR j ← 0 TO KL-1
    i ←   j* 32
    i ←   (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
```

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
```

Intel C/C++ Compiler Intrinsic Equivalent

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);
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);
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);
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);
VPBROADCASTB __m128i _mm_broadcast_si8(char *a);
VPBROADCASTB __m256i _mm256_broadcast_si8(char *a);
VPBROADCASTW __m128i _mm_broadcast_si16(short *a);
VPBROADCASTW __m256i _mm256_broadcast_si16(short *a);
VPBROADCASTD __m128i _mm_broadcast_si32(int *a);
VPBROADCASTD __m256i _mm256_broadcast_si32(int *a);
VPBROADCASTQ __m128i _mm_broadcast_si64(__m64 *a);
VPBROADCASTQ __m256i _mm256_broadcast_si64(__m64 *a);
VPBROADCASTI128 __m256i _mm256_broadcast_si128(__m128d * a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instructions, see Exceptions Type 6; additionally

#UD	If VEX.L = 0 for VPBROADCASTQ, VPBROADCASTI128.

EVEX-encoded instructions, syntax with reg/mem operand, see Exceptions Type E6.

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	RMI	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.NDS.128.66.0F.WIG C2 /r ib VCMPPD xmm1, xmm2, xmm3/m128, imm8	RVMI	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.NDS.256.66.0F.WIG C2 /r ib VCMPPD ymm1, ymm2, ymm3/m256, imm8	RVMI	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.NDS.512.66.0F.W1 C2 /r ib VCMPPD k1 {k2}, zmm2, zmm3/m512/m64bcst{sae}, imm8	FV	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
FV	ModRM:reg (w)	EVEX.vvvv	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 results 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 register. The second 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 (first operand) is an opmask register. Eight comparisons are performed with results 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 (MAX_VL-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 (MAX_VL-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 5-1). 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 5-1). Bits 3 through 7 of the immediate are reserved.

Table 5-1. 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
NGE_UQ	19H	Not-greater-than-or-equal (unordered, nonsignaling)	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

Table 5-1. 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]	
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 5-2. Compiler should treat reserved Imm8 values as illegal syntax.

Table 5-2. 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 5-3, 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 5-3, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. 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 5-3.

Table 5-3. 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) = (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
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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 __m256 _mm256_cmp_pd(__m256 a, __m256 b, int imm)
(V)CMPPD __m128 _mm_cmp_pd(__m128 a, __m128 b, int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 5-1.

Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

CMPPS—Compare Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C2 /r ib CMPPS xmm1, xmm2/m128, imm8	RMI	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.NDS.128.0F.WIG C2 /r ib VCMPPS xmm1, xmm2, xmm3/m128, imm8	RVMI	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.NDS.256.0F.WIG C2 /r ib VCMPPS ymm1, ymm2, ymm3/m256, imm8	RVMI	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.NDS.512.0F.W0 C2 /r ib VCMPPS k1 {k2}, zmm2, zmm3/m512/m32bcst{sae}, imm8	FV	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
FV	ModRM:reg (w)	EVEX.vvvv	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 results 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 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-bit memory location. The destination operand (first operand) is an opmask register. Up to sixteen comparisons are performed with results 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 (MAX_VL-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 (MAX_VL-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 5-1). 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 5-1). 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 5-4. Compiler should treat reserved Imm8 values as illegal syntax.

Table 5-4. Pseudo-Op and CMPPS Implementation

Pseudo-Op	CMPPS Implementation
CMPEQPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 0*
CMPLTPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 1*
CMPLEPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 2*
CMPUNORDPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 3*
CMPNEQPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 4*
CMPNLTPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 5*
CMPNLEPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 6*
CMPORDPS x*mm1, 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 5-5, 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 5-5, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. 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 5-5.

Table 5-5. Pseudo-Op and VCMPPS Implementation

Pseudo-Op	CMPPS Implementation
VCMPEQPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 0*
VCMPLTPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 1*
VCMPLEPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 2*
VCMPUNORDPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 3*
VCMPNEQPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 4*

Table 5-5. Pseudo-Op and VCMPPS Implementation

Pseudo-Op	CMPPS Implementation
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) = (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
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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 __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 5-1.

Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

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	RMI	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.NDS.128.F2.0F.WIG C2 /r ib VCMPSD xmm1, xmm2, xmm3/m64, imm8	RVMI	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.NDS.LIG.F2.0F.W1 C2 /r ib VCMPSD k1 {k2}, xmm2, xmm3/m64{sae}, imm8	T1S	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
T1S	ModRM:reg (w)	EVEX.vvvv	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 results in 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 (MAX_VL-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 (MAX_VL-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 5-1). 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 5-1). 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 5-6. Compiler should treat reserved Imm8 values as illegal syntax.

Table 5-6. Pseudo-Op and CMPSD Implementation

Pseudo-Op	CMPSD Implementation
CMPEQSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 0*
CMPLTSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 1*
CMPLESD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 2*
CMPUNORDSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 3*
CMPNEQSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 4*
CMPNLTSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 5*
CMPNLESD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 6*
CMPORDSD x*mm1, xmm2*	CMPSD x*mm1, 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 5-7, 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 5-7, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. 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 5-7.

Table 5-7. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
VCMPEQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 0*
VCMPLTSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 1*
VCMPLESD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 2*
VCMPUNORDSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 3*
VCMPNEQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 4*
VCMPNLTSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 5*
VCMPNLESD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 6*
VCMPORDSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 7*
VCMPEQ_UQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 8*
VCMPNGESD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 9*
VCMPNGTSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 0AH*
VCMPFALSESD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 0BH*

Table 5-7. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
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[MAX_VL-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[MAX_VL-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 5-1 Denormal.

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.

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	RMI	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.NDS.128.F3.0F.WIG C2 /r ib VCMPSS xmm1, xmm2, xmm3/m32, imm8	RVMI	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.NDS.LIG.F3.0F.W0 C2 /r ib VCMPSS k1 {k2}, xmm2, xmm3/m32{sae}, imm8	T1S	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
T1S	ModRM:reg (w)	EVEX.vvvv	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 results 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 (MAX_VL-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 128:32 of the destination operand are copied from the first source operand. Bits (MAX_VL-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 5-1). 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 5-1). 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 5-8. Compiler should treat reserved Imm8 values as illegal syntax.

Table 5-8. Pseudo-Op and CMPSS Implementation

Pseudo-Op	CMPSS Implementation
CMPEQSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 0*
CMPLTSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 1*
CMPLESS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 2*
CMPUNORDSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 3*
CMPNEQSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 4*
CMPNLTSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 5*
CMPNLESS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 6*
CMPORDSS x*mm1, xmm2*	CMPSS x*mm1, 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 5-7, 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 5-9, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. 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 5-9.

Table 5-9. Pseudo-Op and VCMPSS Implementation

Pseudo-Op	CMPSS Implementation
VCMPEQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 0*
VCMPLTSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 1*
VCMPLESS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 2*
VCMPUNORDSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 3*
VCMPNEQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 4*
VCMPNLTSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 5*
VCMPNLESS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 6*
VCMPORDSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 7*
VCMPEQ_UQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 8*
VCMPNGESS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 9*
VCMPNGTSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 0AH*
VCMPFALSESS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 0BH*

Table 5-9. Pseudo-Op and VCMPSS Implementation

Pseudo-Op	CMPSS Implementation
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[MAX_VL-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[MAX_VL-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 5-1, Denormal.

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.

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	RM	V/V	SSE2	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.128.66.0F.WIG 2F /r VCOMISD xmm1, xmm2/m64	RM	V/V	AVX	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LIG.66.0F.W1 2F /r VCOMISD xmm1, xmm2/m64{sae}	T1S	V/V	AVX512F	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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 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.

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 Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

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
0F 2F /r COMISS xmm1, xmm2/m32	RM	V/V	SSE	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.128.0F.WIG 2F /r VCOMISS xmm1, xmm2/m32	RM	V/V	AVX	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LIG.0F.W0 2F /r VCOMISS xmm1, xmm2/m32{sae}	T1S	V/V	AVX512F	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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 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.

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 Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

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	RM	V/V	SSE2	Divide packed double-precision floating-point values in xmm1 by packed double-precision floating-point values in xmm2/mem.
VEX.NDS.128.66.0F.WIG 5E /r VDIVPD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Divide packed double-precision floating-point values in xmm2 by packed double-precision floating-point values in xmm3/mem.
VEX.NDS.256.66.0F.WIG 5E /r VDIVPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Divide packed double-precision floating-point values in ymm2 by packed double-precision floating-point values in ymm3/mem.
EVEX.NDS.512.66.0F.W1 5E /r VDIVPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Divide packed double-precision floating-point values in zmm2 by packed double-precision FP values in zmm3/m512/m64bcst and write results to zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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 register. The second 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 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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding destination are unmodified.

Operation

VDIVPD (EVEX encoded versions)

```
(KL, VL) = (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);        ; refer to Table 2-1
    ELSE
        SET_RM(MXCSR.RM);
```

```
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← 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
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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 __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 Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

DIVPS—Divide Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 5E /r DIVPS xmm1, xmm2/m128	RM	V/V	SSE	Divide packed single-precision floating-point values in xmm1 by packed single-precision floating-point values in xmm2/mem.
VEX.NDS.128.0F.WIG 5E /r VDIVPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Divide packed single-precision floating-point values in xmm2 by packed single-precision floating-point values in xmm3/mem.
VEX.NDS.256.0F.WIG 5E /r VDIVPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Divide packed single-precision floating-point values in ymm2 by packed single-precision floating-point values in ymm3/mem.
EVEX.NDS.512.0F.W0 5E /r VDIVPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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 register. 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 destination operand is a ZMM 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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VDIVPS (EVEX encoded versions)

```
(KL, VL) = (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
```

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← 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
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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);
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 Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

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	RM	V/V	SSE2	Divide low double-precision floating-point value in xmm1 by low double-precision floating-point value in xmm2/m64.
VEX.NDS.128.F2.0F.WIG 5E /r VDIVSD xmm1, xmm2, xmm3/m64	RVM	V/V	AVX	Divide low double-precision floating-point value in xmm2 by low double-precision floating-point value in xmm3/m64.
EVEX.NDS.LIG.F2.0F.W1 5E /r VDIVSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Divide low double-precision floating-point value in xmm2 by low double-precision floating-point value in xmm3/m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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[MAX_VL-1:128] ← 0
```

VDIVSD (VEX.128 encoded version)

```
DEST[63:0] ←SRC1[63:0] / SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAX_VL-1:128] ←0
```

DIVSD (128-bit Legacy SSE version)

```
DEST[63:0] ←DEST[63:0] / SRC[63:0]
DEST[MAX_VL-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 Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

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	RM	V/V	SSE	Divide low single-precision floating-point value in xmm1 by low single-precision floating-point value in xmm2/m32.
VEX.NDS.128.F3.0F.WIG 5E /r VDIVSS xmm1, xmm2, xmm3/m32	RVM	V/V	AVX	Divide low single-precision floating-point value in xmm2 by low single-precision floating-point value in xmm3/m32.
EVEX.NDS.LIG.F3.0F.W0 5E /r VDIVSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Divide low single-precision floating-point value in xmm2 by low single-precision floating-point value in xmm3/m32.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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[MAX_VL-1:128] ← 0
```

VDIVSS (VEX.128 encoded version)

```
DEST[31:0] ←SRC1[31:0] / SRC2[31:0]
DEST[127:32] ←SRC1[127:32]
DEST[MAX_VL-1:128] ←0
```

DIVSS (128-bit Legacy SSE version)

```
DEST[31:0] ←DEST[31:0] / SRC[31:0]
DEST[MAX_VL-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 Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

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.512.66.0F38.W1 8A /r VCOMPRESSPD zmm1/mV {k1}{z}, zmm2	T1S	V/V	AVX512F	Compress packed double-precision floating-point values from zmm2 using controlmask k1 to zmm1/mV.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Compress (store) up to 8 double-precision floating-point values from the source operand (the second operand) as a contiguous vector to the destination operand (the first operand) The source operand is a ZMM register, the destination operand can be a ZMM register or a 512-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 is ignored.

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) = (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) = (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
```

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);

SIMD Floating-Point Exceptions

None

Other Exceptions

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

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.512.66.0F38.W0 8A /r VCOMPRESSPS zmm1/mV {k1}{z}, zmm2	T1S	V/V	AVX512F	Compress packed single-precision floating-point values from zmm2 using controlmask k1 to zmm1/mV.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Compress (stores) up to 16 single-precision floating-point values from the source operand (the second operand) to the destination operand (the first operand). The source operand is a ZMM register, the destination operand can be a ZMM register or a 512-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 is ignored.

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) = (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) = (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
```

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);

SIMD Floating-Point Exceptions

None

Other Exceptions

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

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	RM	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	RM	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	RM	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed double-precision floating-point values in ymm1.
EVEX.512.F3.0F.W0 E6 /r VCVTDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst	HV	V/V	AVX512F	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
HV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

EVEX encoded versions: The source operand can be 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 vector 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 (MAX_VL-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 (MAX_VL-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.

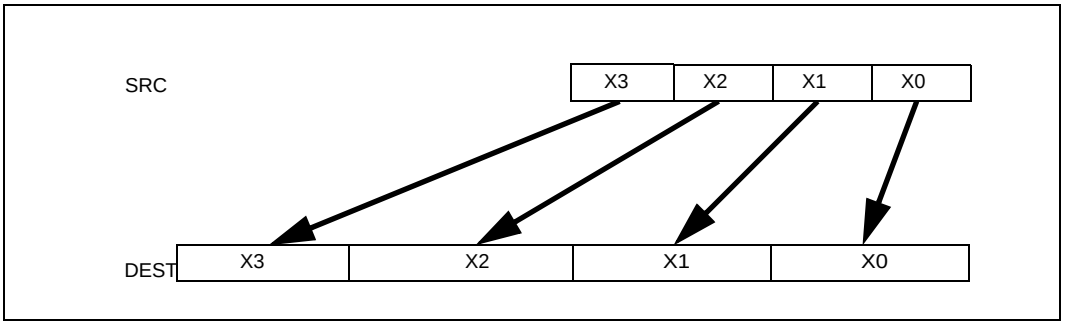


Figure 5-11. CVTDQ2PD (VEX.256 encoded version)

Operation

VCVTDQ2PD (EVEX encoded versions) when src operand is a register

```
(KL, VL) = (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
```

VCVTDQ2PD (EVEX encoded versions) when src operand is a memory source

```
(KL, VL) = (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
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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);
CVTDQ2PD __m256d _mm256_cvtepi32_pd (__m128i src)
CVTDQ2PD __m128d _mm_cvtepi32_pd (__m128i src)

Other Exceptions

VEX-encoded instructions, see Exceptions Type 5; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E5.

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
0F 5B /r CVTDQ2PS xmm1, xmm2/m128	RM	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	RM	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	RM	V/V	AVX	Convert eight packed signed doubleword integers from ymm2/mem to eight packed single-precision floating-point values in ymm1.
EVEX.512.0F.W0 5B /rr VCVTDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed single-precision floating-point values in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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 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. Bits (MAX_VL-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 (MAX_VL-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 (MAX_VL-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) = (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);          ; refer to Table 2-1
    ELSE
        SET_RM(MXCSR.RM);         ; refer to Table 2-1
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
```

VCVTDQ2PS (EVEX encoded versions) when SRC operand is a memory source
(KL, VL) = (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
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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);
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 Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E2.

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	RM	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	RM	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	RM	V/V	AVX	Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1.
EVEX.512.F2.0F.W1 E6 /r VCVTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts packed double-precision floating-point values in the source operand (second operand) to packed 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 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 with writemask k1. The upper bits (MAX_VL-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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-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.

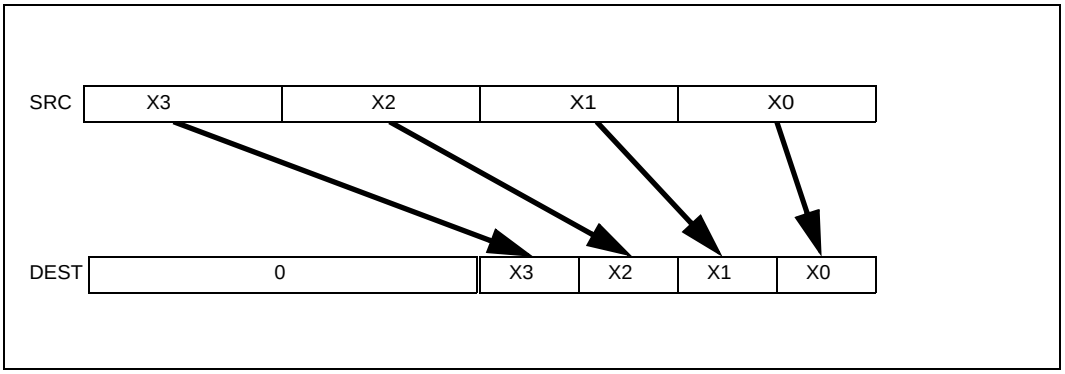


Figure 5-12. VCVTPD2DQ (VEX.256 encoded version)

Operation

VCVTPD2DQ (EVEX encoded versions) when src operand is a register

```
(KL, VL) = (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            Convert_Double_Precision_Floating_Point_To_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[MAX_VL-1:VL/2] ← 0
```

VCVTPD2DQ (EVEX encoded versions) when src operand is a memory source

```
(KL, VL) = (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[MAX_VL-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[MAX_VL-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[MAX_VL-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[MAX_VL-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_cvtpd_epi32 (__m256d src)
CVTPD2DQ __m128i _mm_cvtpd_epi32 (__m128d src)
```

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E2.

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	RM	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	RM	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	RM	V/V	AVX	Convert four packed double-precision floating-point values in ymm2/mem to four single-precision floating-point values in xmm1.
EVEX.512.66.0F.W1 5A /r VCVTPD2PS ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight single-precision floating-point values in ymm1with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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 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 with writemask k1. The upper bits (MAX_VL-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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-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.

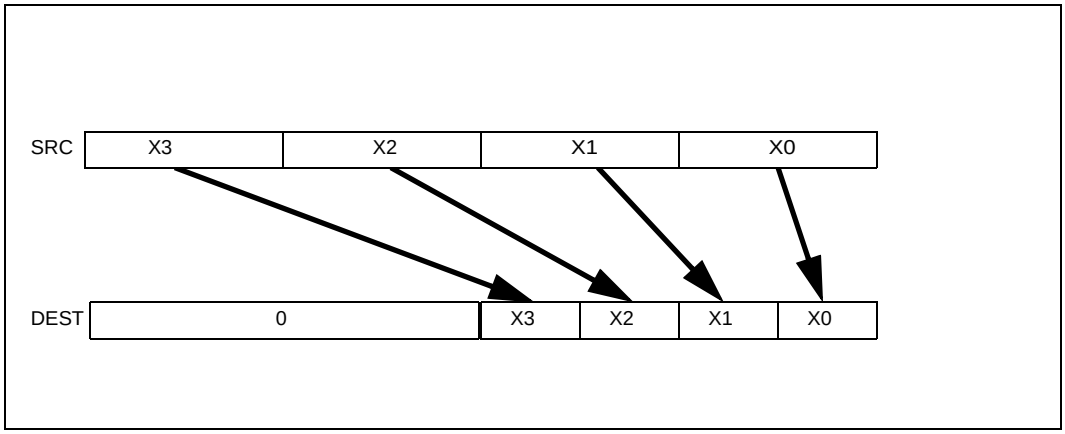


Figure 5-13. VCVTPD2PS (VEX.256 encoded version)

Operation

VCVTPD2PS (EVEX encoded version) when src operand is a register

```
(KL, VL) = (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_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[MAX_VL-1:VL/2] ← 0
```

VCVTPD2PS (EVEX encoded version) when src operand is a memory source

```
(KL, VL) = (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[MAX_VL-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[MAX_VL-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[MAX_VL-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[MAX_VL-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_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 Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E2.

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.512.0F.W1 79 /r VCVTPD2UDQ ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

The source operand is a ZMM 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 with writemask k1. The upper bits (MAX_VL-1:256) of the corresponding destination are zeroed.

Operation

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

```
(KL, VL) =(8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

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

VCVTPD2UDQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (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[MAX_VL-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);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

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

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
HVM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

VEX.128 version: The source operand is a XMM register or 64-bit memory location. The destination operand is a XMM register. The upper bits (MAX_VL-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 (MAX_VL-1:256) of the corresponding destination register are zeroed.

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

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

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

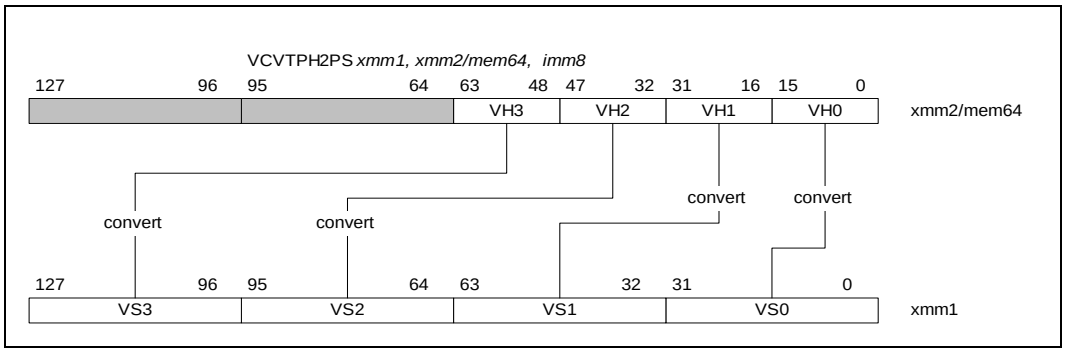


Figure 5-14. VCVTPH2PS (128-bit Version)

Operation

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

VCVTPH2PS (EVEX encoded versions)

```
(KL, VL) = (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
```

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[MAX_VL-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[MAX_VL-1:128] ← 0
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2PS __m512 _mm512_cvtph_ps(__m256i a);
VCVTPH2PS __m512 _mm512_mask_cvtph_ps(__m512 s, __mmask16 k, __m256i a);
VCVTPH2PS __m512 _mm512_maskz_cvtph_ps(__mmask16 k, __m256i a);
VCVTPH2PS __m512 _mm512_cvt_roundph_ps(__m256i a, int sae);
VCVTPH2PS __m512 _mm512_mask_cvt_roundph_ps(__m512 s, __mmask16 k, __m256i a, int sae);
VCVTPH2PS __m512 _mm512_maskz_cvt_roundph_ps(__mmask16 k, __m256i a, int sae);
VCVTPH2PS __m128 _mm_cvtph_ps (__m128i m1);

VCVTPH2PS __m256 _mm256_cvtph_ps (__m128i m1)

SIMD Floating-Point Exceptions

Invalid

Other Exceptions

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

#UD	If VEX.W=1.

EVEX-encoded instructions, see Exceptions Type E11.

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	MRI	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	MRI	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.512.66.0F3A.W0 1D /r ib VCVTPS2PH ymm1/m256 {k1}{z}, zmm2{sae}, imm8	HVM	V/V	AVX512F	Convert sixteen packed single-precision floating-point values in zmm2 to packed half-precision (16-bit) floating-point values in ymm1/m256. Imm8 provides rounding controls.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
HVM	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.

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 (MAX_VL-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 (MAX_VL-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 register. The destination operand is a YMM register or a 256-bit memory location, conditionally updated with writemask k1. Bits (MAX_VL-1:256) of the corresponding destination register are zeroed.

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

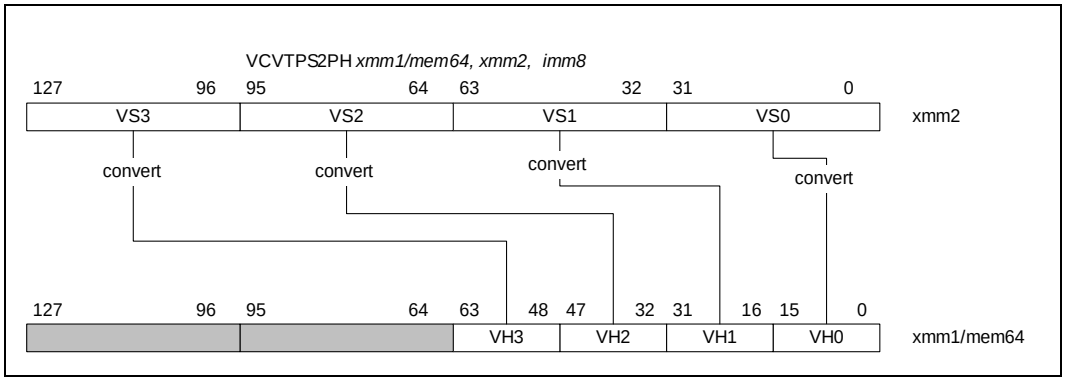


Figure 5-15. VCVTPS2PH (128-bit Version)

The immediate byte defines several bit fields that control rounding operation. The effect and encoding of the RC field are listed in Table 5-10.

Table 5-10. 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	

Operation

```
vCvt_s2h(SRC1[31:0])
{
IF Imm[2] = 0
THEN    ; using Imm[1:0] for rounding control, see Table 5-10
    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) = (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[MAX_VL-1:VL/2] ← 0
```

VCVTPS2PH (EVEX encoded versions) when dest is memory

```
(KL, VL) = (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[MAX_VL-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[MAX_VL-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 _mm_cvtps_ph (__m128 m1, const int imm);
VCVTPS2PH __m128i _mm256_cvtps_ph(__m256 m1, const int imm);

SIMD Floating-Point Exceptions

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

Other Exceptions

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

#UD	If VEX.W=1.

EVEX-encoded instructions, see Exceptions Type E11NF.

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	RM	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	RM	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	RM	V/V	AVX	Convert eight packed single-precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1.
EVEX.512.66.0F.W0 5B /r VCVTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-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) = (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
```

```
    ELSE
        SET_RM(MXCSR.RM);
FI;

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

VCVTPS2DQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (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
```

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[MAX_VL-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[MAX_VL-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_cvtps_epi32 (__m256 a)
CVTPS2DQ __m128i _mm_cvtps_epi32 (__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E2.

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.512.0F.W0 79 /r VCVTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

The source operand is a ZMM 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.

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) = (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

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

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

```
(KL, VL) = (16, 512)
```

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

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);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

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
0F 5A /r CVTPS2PD xmm1, xmm2/m64	RM	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	RM	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	RM	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.512.0F.W0 5A /r VCVTPS2PD zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}	HV	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
HV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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 register, a 256-bit memory location or a 256-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 an XMM register or 128- bit memory location. The destination operand is a YMM register. Bits (MAX_VL-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 (MAX_VL-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 (MAX_VL-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.

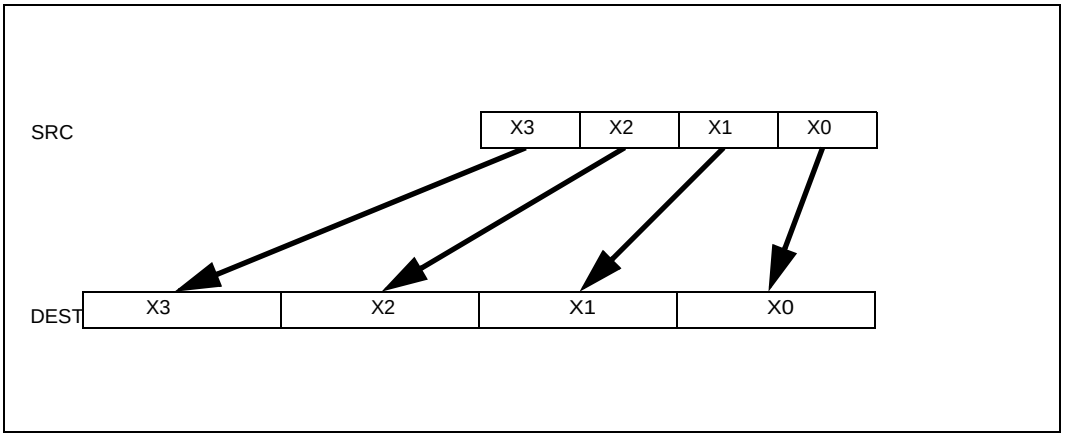


Figure 5-16. CVTPS2PD (VEX.256 encoded version)

Operation

VCVTPS2PD (EVEX encoded versions) when src operand is a register

```
(KL, VL) = (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
```

VCVTPS2PD (EVEX encoded versions) when src operand is a memory source

```
(KL, VL) = (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
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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);
CVTPS2PD __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 Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3.

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	RM	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	RM	V/N.E.	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
VEX.128.F2.0F.W0 2D /r VCVTSD2SI r32, xmm1/m64	RM	V/V	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
VEX.128.F2.0F.W1 2D /r VCVTSD2SI r64, xmm1/m64	RM	V/N.E.[1]	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
EVEX.LIG.F2.0F.W0 2D /r VCVTSD2SI r32, xmm1/m64{er}	T1F	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
EVEX.LIG.F2.0F.W1 2D /r VCVTSD2SI r64, xmm1/m64{er}	T1F	V/N.E.[1]	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a double-precision floating-point value in the source operand (the second operand) to a signed double-word 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 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 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.

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 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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode and OperandSize = 64
    THEN    DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_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 Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F2.0F.W0 79 /r VCVTSD2USI r32, xmm1/m64{er}	T1F	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32.
EVEX.LIG.F2.0F.W1 79 /r VCVTSD2USI r64, xmm1/m64{er}	T1F	V/N.E.[1]	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64.

NOTES:

1. Encoding the VEX prefix VEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

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

Operation

VCVTSD2USI (EVEX encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSD2USI unsigned int _mm_cvtsd_u32(__m128d);
VCVTSD2USI unsigned int _mm_cvt_roundsd_u32(__m128d, int r);
VCVTSD2USI unsigned int __int64 _mm_cvtsd_u64(__m128d);
VCVTSD2USI unsigned int __int64 _mm_cvt_roundsd_u64(__m128d, int r);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3NF.

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	RM	V/V	SSE2	Convert one double-precision floating-point value in xmm2/m64 to one single-precision floating-point value in xmm1.
VEX.NDS.128.F2.0F.WIG 5A /r VCVTSD2SS xmm1,xmm2, xmm3/m64	RVM	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.NDS.LIG.F2.0F.W1 5A /r VCVTSD2SS xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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[MAX_VL-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[MAX_VL-1:128] ←0
```

CVTSD2SS (128-bit Legacy SSE version)

```
DEST[31:0] ←Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[MAX_VL-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 Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

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	RM	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	RM	V/N.E.	SSE2	Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.
VEX.NDS.128.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	RVM	V/V	AVX	Convert one signed doubleword integer from r/m32 to one double-precision floating-point value in xmm1.
VEX.NDS.128.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64	RVM	V/N.E.[1]	AVX	Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.
EVEX.NDS.LIG.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	T1S	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one double-precision floating-point value in xmm1.
EVEX.NDS.LIG.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64{er}	T1S	V/N.E.[1]	AVX512F	Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-1:128) of the destination register are zeroed.

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

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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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[MAX_VL-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[MAX_VL-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[MAX_VL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSI2SD __m128d _mm_cvti32_sd(__m128d s, int a);
VCVTSI2SD __m128d _mm_cvt_roundi32_sd(__m128d s, int a, int r);
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 Exceptions Type 3 if W1, else Type 5.

EVEX-encoded instructions, see Exceptions Type E3NF if W1, else Type E10NF.

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	RM	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	RM	V/N.E.	SSE	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.
VEX.NDS.128.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32	RVM	V/V	AVX	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
VEX.NDS.128.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64	RVM	V/N.E.[1]	AVX	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.
EVEX.NDS.LIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32{er}	T1S	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
EVEX.NDS.LIG.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64{er}	T1S	V/N.E.[1]	AVX512F	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-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.

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

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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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[MAX_VL-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[MAX_VL-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[MAX_VL-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 Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3NF.

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	RM	V/V	SSE2	Convert one single-precision floating-point value in xmm2/m32 to one double-precision floating-point value in xmm1.
VEX.NDS.128.F3.0F.WIG 5A /r VCVTSS2SD xmm1, xmm2, xmm3/m32	RVM	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.NDS.LIG.F3.0F.W0 5A /r VCVTSS2SD xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	T1S	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-1:128) of the destination register are zeroed.

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

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[MAX_VL-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[MAX_VL-1:128] ←0

CVTSS2SD (128-bit Legacy SSE version)
DEST[63:0] ←Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[MAX_VL-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 Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.

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	RM	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	RM	V/N.E.	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
VEX.128.F3.0F.W0 2D /r VCVTSS2SI r32, xmm1/m32	RM	V/V	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
VEX.128.F3.0F.W1 2D /r VCVTSS2SI r64, xmm1/m32	RM	V/N.E.[1]	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
EVEX.LIG.F3.0F.W0 2D /r VCVTSS2SI r32, xmm1/m32{er}	T1F	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
EVEX.LIG.F3.0F.W1 2D /r VCVTSS2SI r64, xmm1/m32{er}	T1F	V/N.E.[1]	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a single-precision floating-point value in the source operand (the second operand) to 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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] ← Convert_Single_Precision_Floating_Point_To_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 Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F3.0F.W0 79 /r VCVTSS2USI r32, xmm1/m32{er}	T1F	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32.
EVEX.LIG.F3.0F.W1 79 /r VCVTSS2USI r64, xmm1/m32{er}	T1F	V/N.E.[1]	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

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

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

Operation

VCVTSS2USI (EVEX encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2USI unsigned _mm_cvtss_u32(__m128 a);
VCVTSS2USI unsigned _mm_cvt_roundss_u32(__m128 a, int r);
VCVTSS2USI unsigned __int64 _mm_cvtss_u64(__m128 a);
VCVTSS2USI unsigned __int64 _mm_cvt_roundss_u64(__m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3NF.

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	RM	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	RM	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	RM	V/V	AVX	Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1 using truncation.
EVEX.512.66.0F.W1 E6 /r VCVTTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst{sae}	FV	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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 register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register conditionally updated with writemask k1. The upper bits (MAX_VL-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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-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.

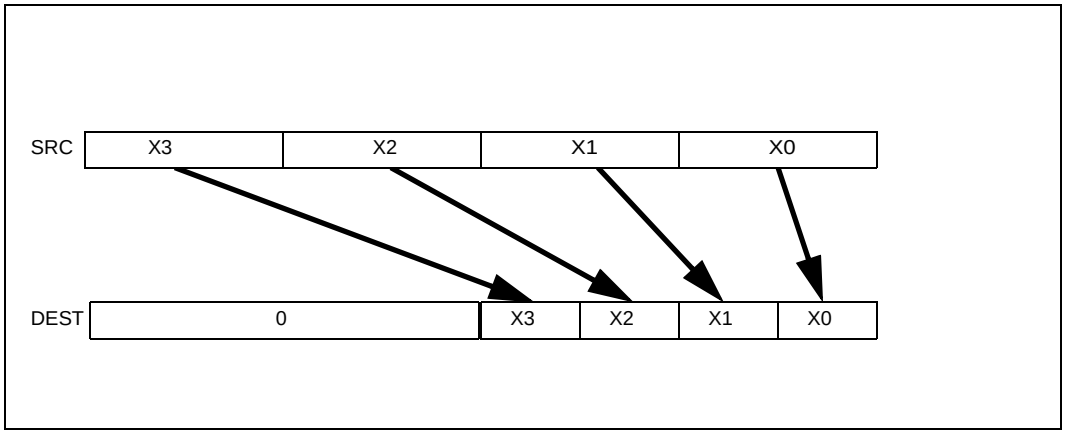


Figure 5-17. VCVTTPD2DQ (VEX.256 encoded version)

Operation

VCVTTPD2DQ (EVEX encoded versions) when src operand is a register
(KL, VL) = (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[MAX_VL-1:VL/2] ← 0
```

VCVTTPD2DQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (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[MAX_VL-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[MAX_VL-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[MAX_VL-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[MAX_VL-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_cvttpd_epi32 (__m256d src);
CVTTPD2DQ __m128i _mm_cvttpd_epi32 (__m128d src);
```

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E2.

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.512.0F.W1 78 /r VCVTTPD2UDQ ymm1 {k1}{z}, zmm2/m512/m64bcst{sae}	FV	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register 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 register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register conditionally updated with writemask k1. The upper bits (MAX_VL-1:256) of the corresponding destination are zeroed.

Operation

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

```
(KL, VL) = (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[MAX_VL-1:VL/2] ← 0
```

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

```
(KL, VL) = (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[MAX_VL-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);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

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	RM	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	RM	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	RM	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.512.F3.0F.W0 5B /r VCVTTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	FV	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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 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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-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) = (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
```

VCVTTPS2DQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (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
```

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[MAX_VL-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[MAX_VL-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_cvttps_epi32 (__m256 a)
CVTTPS2DQ __m128i _mm_cvttps_epi32 (__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E2.

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.512.0F.W0 78 /r VCVTTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}	FV	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

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.

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) = (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
```

VCVTTPS2UDQ (EVEX encoded versions) when src operand is a memory source

```
(KL, VL) = (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
```

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);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

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	RM	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	RM	V/N.E.	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
VEX.128.F2.0F.W0 2C /r VCVTTSD2SI r32, xmm1/m64	RM	V/V	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
VEX.128.F2.0F.W1 2C /r VCVTTSD2SI r64, xmm1/m64	T1F	V/N.E.[1]	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
EVEX.LIG.F2.0F.W0 2C /r VCVTTSD2SI r32, xmm1/m64{sae}	T1F	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
EVEX.LIG.F2.0F.W1 2C /r VCVTTSD2SI r64, xmm1/m64{sae}	T1F	V/N.E.[1]	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a double-precision floating-point value in the source operand (the second operand) to a signed double-word 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, 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 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 Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F2.0F.W0 78 /r VCVTTSD2USI r32, xmm1/m64{sae}	T1F	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32 using truncation.
EVEX.LIG.F2.0F.W1 78 /r VCVTTSD2USI r64, xmm1/m64{sae}	T1F	V/N.E.[1]	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64 using truncation.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

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 Exceptions Type E3NF.

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	RM	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	RM	V/N.E.	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
VEX.128.F3.0F.W0 2C /r VCVTTSS2SI r32, xmm1/m32	RM	V/V	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
VEX.128.F3.0F.W1 2C /r VCVTTSS2SI r64, xmm1/m32	RM	V/N.E.[1]	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
EVEX.LIG.F3.0F.W0 2C /r VCVTTSS2SI r32, xmm1/m32{sae}	T1F	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
EVEX.LIG.F3.0F.W1 2C /r VCVTTSS2SI r64, xmm1/m32{sae}	T1F	V/N.E.[1]	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a single-precision floating-point value in the source operand (the second operand) to a signed double-word 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 Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F3.0F.W0 78 /r VCVTTSS2USI r32, xmm1/m32{sae}	T1F	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32 using truncation.
EVEX.LIG.F3.0F.W1 78 /r VCVTTSS2USI r64, xmm1/m32{sae}	T1F	V/N.E.[1]	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64 using truncation.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

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

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

Operation

VCVTTSS2USI (EVEX encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSS2USI unsigned int _mm_cvttss_u32(__m128 a);
VCVTTSS2USI unsigned int _mm_cvtt_roundss_u32(__m128 a, int sae);
VCVTTSS2USI unsigned __int64 _mm_cvttss_u64(__m128 a);
VCVTTSS2USI unsigned __int64 _mm_cvtt_roundss_u64(__m128 a, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3NF.

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

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
HV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

The source operand is a YMM register, a 256-bit memory location or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM 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) = (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
```

VCVTUDQ2PD (EVEX encoded versions) when src operand is a memory source

```
(KL, VL) = (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
```

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);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E5.

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.512.F2.0F.W0 7A /r VCVTUDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to sixteen packed single-precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

The source operand is a ZMM 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.

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) = (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

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

VCVTUDQ2PS (EVEX encoded version) when src operand is a memory source

```
(KL, VL) = (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
```

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);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

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.NDS.LIG.F2.0F.W0 7B /r VCVTUSI2SD xmm1, xmm2, r/m32	T1S	V/V	AVX512F	Convert one unsigned doubleword integer from r/m32 to one double-precision floating-point value in xmm1.
EVEX.NDS.LIG.F2.0F.W1 7B /r VCVTUSI2SD xmm1, xmm2, r/m64{er}	T1S	V/N.E.[1]	AVX512F	Convert one unsigned quadword integer from r/m64 to one double-precision floating-point value in xmm1.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the second source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

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

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

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

Operation

```
VCVTUSI2SD (EVEX encoded version)
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] ← Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] ← Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-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 Exceptions Type E3NF if W1, else type E10NF.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.F3.0F.W0 7B /r VCVTUSI2SS xmm1, xmm2, r/m32{er}	T1S	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
EVEX.NDS.LIG.F3.0F.W1 7B /r VCVTUSI2SS xmm1, xmm2, r/m64{er}	T1S	V/N.E.[1]	AVX512F	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.

NOTES:

1. Encoding the VEX prefix VEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

Description

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

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

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

Operation

```
VCVTUSI2SS (EVEX encoded version)
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] ← Convert_UInteger_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] ← Convert_UInteger_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-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 Exceptions Type E3NF.

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.512.66.0F38.W1 88 /r VEXPANDPD zmm1 {k1}{z}, zmm2/m512	T1S	V/V	AVX512F	Expand packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Expand (load) up to 8, 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 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 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) = (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
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VEXPANDPD __m512d _mm512_expand_pd( __m512d a);
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);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.nb.

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.512.66.0F38.W0 88 /r VEXPANDPS zmm1 {k1}{z}, zmm2/mV	T1S	V/V	AVX512F	Expand packed single-precision floating-point values from zmm2/memory to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Expand (load) up to 16, 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 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 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) = (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
```

Intel C/C++ Compiler Intrinsic Equivalent

VEXPANDPS __m512 _mm512_expand_ps(__m512 a);
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);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.nb.

VEXTRACTF128/VEXTRACTF32x4/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	RMI	V/V	AVX	Extract 128 bits of packed floating-point values from ymm2 and store results in xmm1/m128.
EVEX.512.66.0F3A.W0 19 /r ib VEXTRACTF32x4 xmm1/m128 {k1}{z}, zmm2, imm8	T4	V/V	AVX512F	Extract 128 bits of packed single-precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W1 1B /r ib VEXTRACTF64x4 ymm1/m256 {k1}{z}, zmm2, imm8	T4	V/V	AVX512F	Extract 256 bits of packed double-precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
T4	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

Description

VEXTRACTF128/VEXTRACTF32x4 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[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.

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] 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 = 512
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[MAX_VL-1:128] ← 0
```

VEXTRACTF32x4 (EVEX encoded versions) when destination is memory

```
VL = 512
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
```

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[MAX_VL-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[MAX_VL-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);
VEXTRACTI32x4 __m128 _mm512_maskz_extractf32x4_ps(__mmask8 k, __m512 a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_extractf64x4_pd(__m512d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_mask_extractf64x4_pd(__m256d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_maskz_extractf64x4_pd(__mmask8 k, __m512d a, const int nidx);
VEXTRACTF128 __m128 _mm256_extractf128_ps (__m256 a, int offset);
VEXTRACTF128 __m128d _mm256_extractf128_pd (__m256d a, int offset);
VEXTRACTF128 __m128i_mm256_extractf128_si256(__m256i a, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions, see Exceptions Type 6; additionally

#UD	IF VEX.L = 0.

EVEX-encoded instructions, see Exceptions Type E6NF.

VEXTRACTI128/VEXTRACTI32x4/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	RMI	V/V	AVX2	Extract 128 bits of integer data from ymm2 and store results in xmm1/m128.
EVEX.512.66.0F3A.W0 39 /r ib VEXTRACTI32x4 xmm1/m128 {k1}{z}, zmm2, imm8	T4	V/V	AVX512F	Extract 128 bits of double-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W1 3B /r ib VEXTRACTI64x4 ymm1/m256 {k1}{z}, zmm2, imm8	T4	V/V	AVX512F	Extract 256 bits of quad-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
T4	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

Description

VEXTRACTI128/VEXTRACTI32x4 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[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.

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] as the multiply factor The destination may be either a vector register or a 256-bit memory location.

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 = 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.
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[MAX_VL-1:128] ← 0
```

VEXTRACTI32x4 (EVEX encoded versions) when destination is memory

```
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.

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
```

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[MAX_VL-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[MAX_VL-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);
VEXTRACTI64x4 __m256i _mm512_extracti64x4_epi64(__m512i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_mask_extracti64x4_epi64(__m256i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_maskz_extracti64x4_epi64(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI128 __m128i _mm256_extracti128_si256(__m256i a, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions, see Exceptions Type 6; additionally

#UD	IF VEX.L = 0.

EVEX-encoded instructions, see Exceptions Type E6NF.

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	RMI	V/V	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	RMI	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 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8	T1S	V/V	**AVX512F**	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
T1S	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

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 Exceptions Type 5; Additionally

#UD	IF VEX.L = 1.

EVEX-encoded instructions, see Exceptions Type E9NF.

VFIXUPIMMPD—Fix Up Special Packed Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F3A.W1 54 /r ib VFIXUPIMMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}, imm8	FV	V/V	AVX512F	Fix up elements of float64 vector in zmm2 using int64 vector table in zmm3/m512/m64bcst, combine with preserved elements from zmm1, and store the result in zmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

Perform fix-up of quad-word elements encoded in double-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand). The destination and the first source operands are ZMM registers. The second 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 two-level look-up table perform a fix-up of each DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

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

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

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

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 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 (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] unmodified;              ; 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
    ; faults are mutually exclusive and TOKENs priority defines the order
    CASE(tsrc[31:0] of TOKEN_TYPE) {
        ZERO_VALUE_TOKEN: if imm8[0] then set #ZE;
        ZERO_VALUE_TOKEN: if imm8[1] then set #IE;
        ONE_VALUE_TOKEN: if imm8[2] then set #ZE;
        ONE_VALUE_TOKEN: if imm8[3] then set #IE;
        SNAN_TOKEN:if imm8[4] then set #IE;
        NEG_INF_TOKEN: if imm8[5] then set #IE;
        NEG_VALUE_TOKEN: if imm8[6] then set #IE;
        POS_INF_TOKEN: if imm8[7] then set #IE;
    }    ; end fault reporting CASE

}        ; end of FIXUPIMM_DP()
```

VFIXUPIMMPD
(KL, VL) = (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(SRC1[i+63:i], SRC2[63:0], imm8 [7:0])
                ELSE
                    DEST[i+63:i] ← FIXUPIMM_DP(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
```

Immediate Control Description:

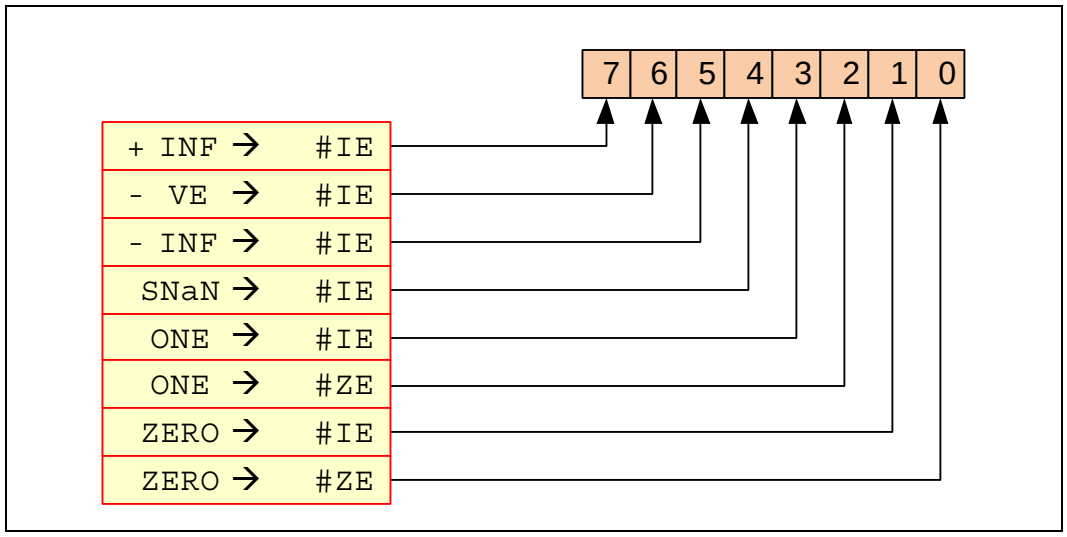


Figure 5-18. VFIXUPIMMPD Immediate Control Description

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);

SIMD Floating-Point Exceptions

Zero, Invalid

Other Exceptions

See Exceptions Type E2.

VFIXUPIMMPS—Fix Up Special Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F3A.W0 54 /r ib VFIXUPIMMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}, imm8	FV	V/V	AVX512F	Fix up elements of float32 vector in zmm2 using int32 vector table in zmm3/m512/m32bcst, combine with preserved elements from zmm1, and store the result in zmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

Perform fix-up of doubleword elements encoded in single-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand). The destination and the first source operands are ZMM 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 two-level look-up table perform a fix-up of each SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

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

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

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

MXCSR.DAZ is used and refer to zmm2 only (i.e. zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}
```

```
FIXUPIMM_SP (src1[31:0],tbl3[63:0], 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] unmodified;             ; 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
    ; faults are mutually exclusive and TOKENs priority defines the order
    CASE(tsrc[31:0] of TOKEN_TYPE) {
        ZERO_VALUE_TOKEN: if imm8[0] then set #ZE;
        ZERO_VALUE_TOKEN: if imm8[1] then set #IE;
        ONE_VALUE_TOKEN: if imm8[2] then set #ZE;
        ONE_VALUE_TOKEN: if imm8[3] then set #IE;
        SNAN_TOKEN:if imm8[4] then set #IE;
        NEG_INF_TOKEN: if imm8[5] then set #IE;
        NEG_VALUE_TOKEN: if imm8[6] then set #IE;
        POS_INF_TOKEN: if imm8[7] then set #IE;
    }           ; end fault reporting CASE

}       ; end of FIXUPIMM_SP()
```

VFIXUPIMMPS (EVEX)
(KL, VL) = (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(SRC1[i+31:i], SRC2[31:0], imm8 [7:0])
                ELSE
                    DEST[i+31:i] ← FIXUPIMMPS(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
```

Immediate Control Description:

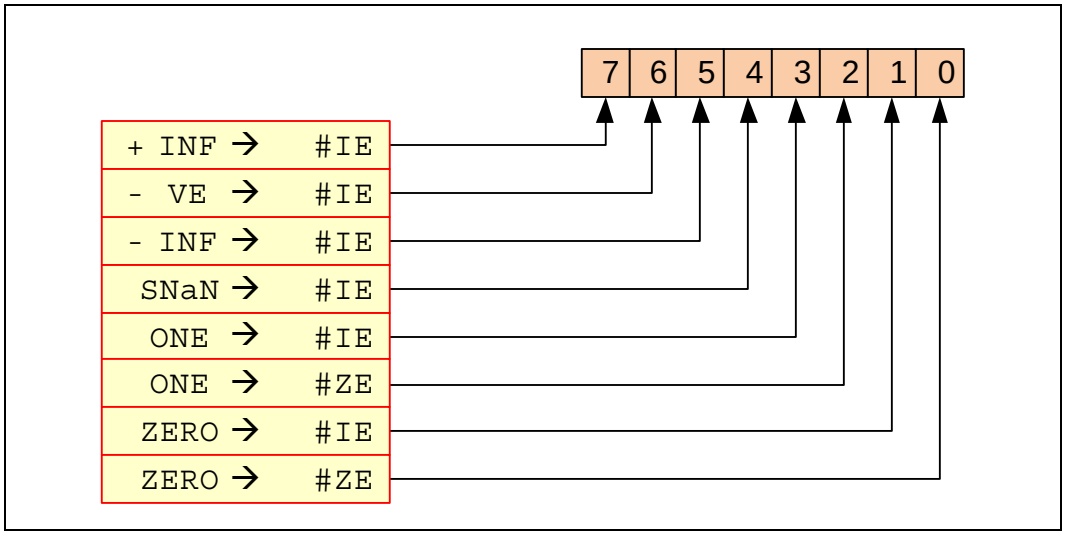


Figure 5-19. VFIXUPIMMPS Immediate Control Description

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);

SIMD Floating-Point Exceptions

Zero, Invalid

Other Exceptions

See Exceptions Type E2.

VFIXUPIMMSD—Fix Up Special Scalar Float64 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 55 /r ib VFIXUPIMMSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	T1S	V/V	AVX512F	Fix up a float64 number in the low quadword element of xmm2 using scalar int32 table in xmm3/m64 and store the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

Perform a fix-up of the low quadword element encoded in double-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 64- bit memory location.

The two-level look-up table perform a fix-up of each DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, 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 (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] unmodified;             ; 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
    ; faults are mutually exclusive and TOKENs priority defines the order
    CASE(tsrc[31:0] of TOKEN_TYPE) {
        ZERO_VALUE_TOKEN: if imm8[0] then set #ZE;
        ZERO_VALUE_TOKEN: if imm8[1] then set #IE;
        ONE_VALUE_TOKEN: if imm8[2] then set #ZE;
        ONE_VALUE_TOKEN: if imm8[3] then set #IE;
        SNAN_TOKEN:if imm8[4] then set #IE;
        NEG_INF_TOKEN: if imm8[5] then set #IE;
        NEG_VALUE_TOKEN: if imm8[6] then set #IE;
        POS_INF_TOKEN: if imm8[7] then set #IE;
    }           ; end fault reporting CASE

}       ; end of FIXUPIMM_DP()
```

VFIXUPIMMSD (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[63:0] ← FIXUPIMM_PD(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[MAX_VL-1:128] ← 0
```

Immediate Control Description:

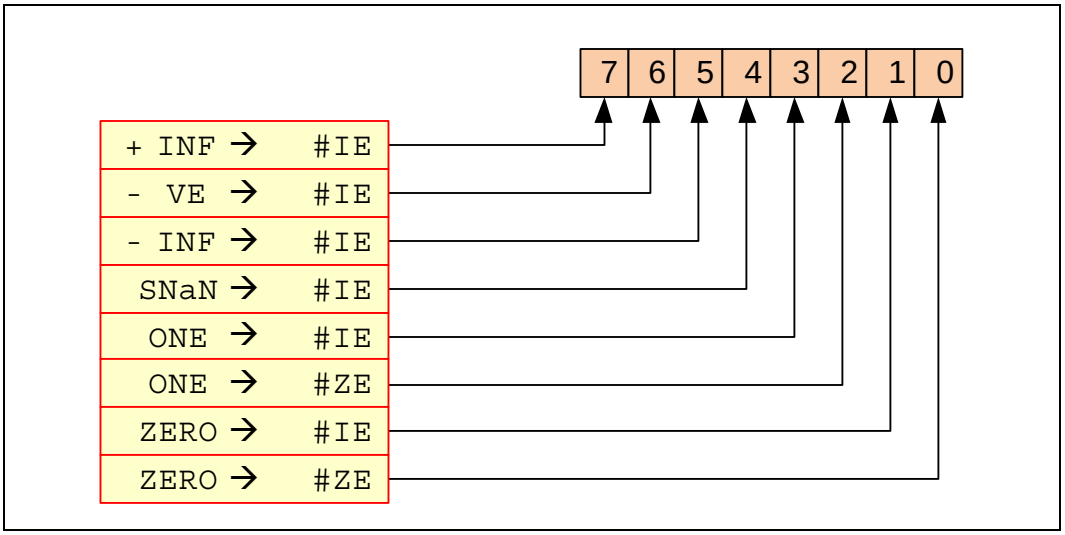


Figure 5-20. VFIXUPIMMSD Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Zero, Invalid

Other Exceptions

See Exceptions Type E3.

VFIXUPIMMSS—Fix Up Special Scalar Float32 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 55 /r ib VFIXUPIMMSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	T1S	V/V	AVX512F	Fix up a float32 number in the low doubleword element in xmm2 using scalar int32 table in xmm3/m32 and store the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

Perform a fix-up of the low doubleword element encoded in single-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low doubleword element of the destination operand (the first operand) Bits 127:32 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 32-bit memory location.

The two-level look-up table perform a fix-up of each SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=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 (src1[31:0],tbl3[63:0], 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] unmodified;             ; 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
    ; faults are mutually exclusive and TOKENs priority defines the order
    CASE(tsrc[31:0] of TOKEN_TYPE) {
        ZERO_VALUE_TOKEN: if imm8[0] then set #ZE;
        ZERO_VALUE_TOKEN: if imm8[1] then set #IE;
        ONE_VALUE_TOKEN: if imm8[2] then set #ZE;
        ONE_VALUE_TOKEN: if imm8[3] then set #IE;
        SNAN_TOKEN:if imm8[4] then set #IE;
        NEG_INF_TOKEN: if imm8[5] then set #IE;
        NEG_VALUE_TOKEN: if imm8[6] then set #IE;
        POS_INF_TOKEN: if imm8[7] then set #IE;
    }           ; end fault reporting CASE

}       ; end of FIXUPIMM_SP()
```

VFIXUPIMMSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← FIXUPIMM_SP(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[MAX_VL-1:128] ← 0
```

Immediate Control Description:

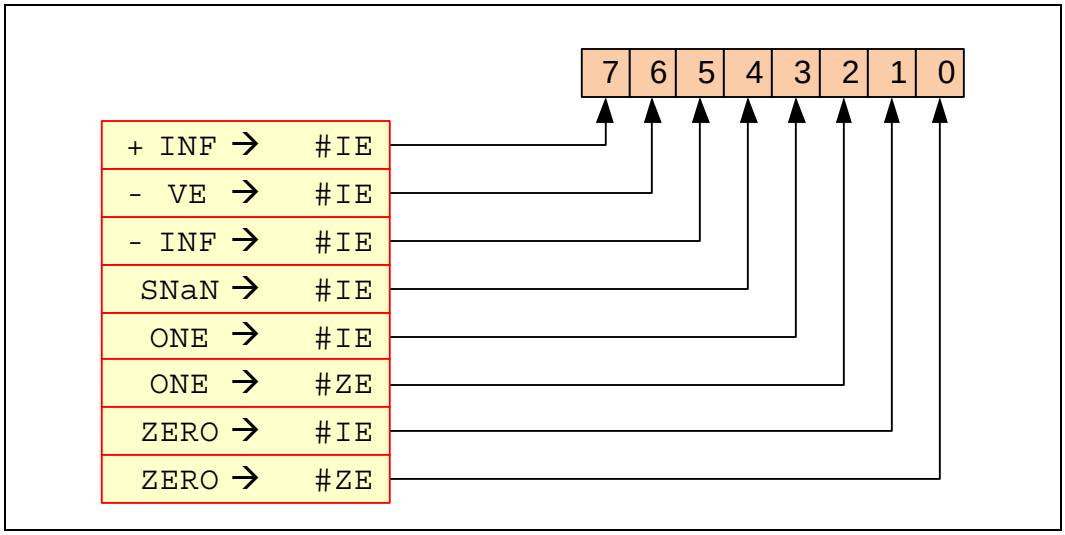


Figure 5-21. VFIXUPIMMSS Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Zero, Invalid

Other Exceptions

See Exceptions Type E3.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W1 98 /r VFMADD132PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.NDS.128.66.0F38.W1 A8 /r VFMADD213PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, add to xmm2/mem and put result in xmm0.
VEX.NDS.128.66.0F38.W1 B8 /r VFMADD231PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.
VEX.NDS.256.66.0F38.W1 98 /r VFMADD132PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, add to ymm1 and put result in ymm0.
VEX.NDS.256.66.0F38.W1 A8 /r VFMADD213PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, add to ymm2/mem and put result in ymm0.
VEX.NDS.256.66.0F38.W1 B8 /r VFMADD231PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, add to ymm0 and put result in ymm0.
EVEX.NDS.512.66.0F38.W1 98 /r VFMADD132PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm0 and zmm2/m512/m64bcst, add to zmm1 and put result in zmm0.
EVEX.NDS.512.66.0F38.W1 A8 /r VFMADD213PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm0 and zmm1, add to zmm2/m512/m64bcst and put result in zmm0.
EVEX.NDS.512.66.0F38.W1 B8 /r VFMADD231PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2/m512/m64bcst, add to zmm0 and put result in zmm0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

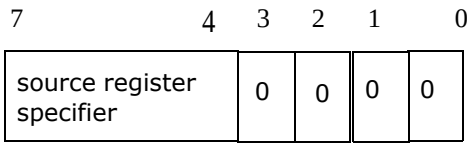
Performs a set of SIMD multiply-add computation on packed double-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFMADD213PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFMADD231PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

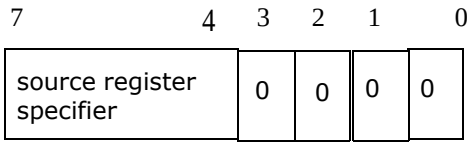
EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in



Bits[3:0]: Reserved for Future Use and must be Zero

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



Bits[3:0]: Reserved for Future Use and must be Zero

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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

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

```
(KL, VL) = (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
```

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

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

```
(KL, VL) = (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
```

VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a is a register)

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

```
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
```

VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (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
```

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

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

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

```
(KL, VL) = (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
```

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);
VFMADD132PD __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);
VFMADD213PD __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);
VFMADD231PD __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);
VFMADD132PD __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);
VFMADD213PD __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);
VFMADD231PD __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 98 /r VFMADD132PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.NDS.128.66.0F38.W0 A8 /r VFMADD213PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, add to xmm2/mem and put result in xmm0.
VEX.NDS.128.66.0F38.W0 B8 /r VFMADD231PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.
VEX.NDS.256.66.0F38.W0 98 /r VFMADD132PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, add to ymm1 and put result in ymm0.
VEX.NDS.256.66.0F38.W0 A8 /r VFMADD213PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, add to ymm2/mem and put result in ymm0.
VEX.NDS.256.66.0F38.0 B8 /r VFMADD231PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, add to ymm0 and put result in ymm0.
EVEX.NDS.512.66.0F38.W0 98 /r VFMADD132PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm0 and zmm2/m512/m32bcst, add to zmm1 and put result in zmm0.
EVEX.NDS.512.66.0F38.W0 A8 /r VFMADD213PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm0 and zmm1, add to zmm2/m512/m32bcst and put result in zmm0.
EVEX.NDS.512.66.0F38.W0 B8 /r VFMADD231PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2/m512/m32bcst, add to zmm0 and put result in zmm0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

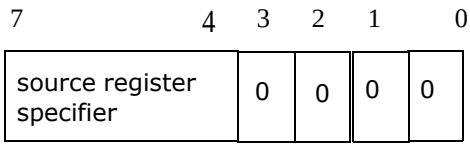
Performs a set of SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

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

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

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

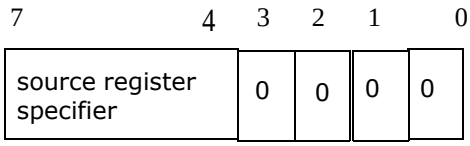
EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in



Bits[3:0]: Reserved for Future Use and must be Zero

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



Bits[3:0]: Reserved for Future Use and must be Zero

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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
```

```
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
```

VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (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
```

VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

```
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
```

VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (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
```

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

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

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

```
(KL, VL) = (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
```

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);
VFMADD132PS __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);
VFMADD213PS __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);
VFMADD231PS __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);
VFMADD132PS __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);
VFMADD213PS __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);
VFMADD231PS __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 99 /r VFMADD132SD xmm0, xmm1, xmm2/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm2/m64, add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 A9 /r VFMADD213SD xmm0, xmm1, xmm2/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm1, add to xmm2/m64 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 B9 /r VFMADD231SD xmm0, xmm1, xmm2/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2/m64, add to xmm0 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W1 99 /r VFMADD132SD xmm0 {k1}{z}, xmm1, xmm2/m64{er}	F1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm0 and xmm2/m64, add to xmm1 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm0 {k1}{z}, xmm1, xmm2/m64{er}	F1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm0 and xmm1, add to xmm2/m64 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm0 {k1}{z}, xmm1, xmm2/m64{er}	F1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2/m64, add to xmm0 and put result in xmm0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD multiply-add computation on the low packed 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.

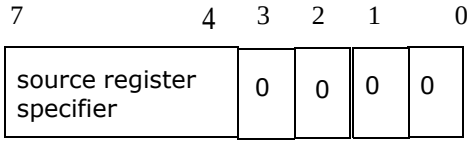
VFMADD132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMADD213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMADD231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the 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.



Bits[3:0]: Reserved for Future Use and must be Zero

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

VFMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← 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[MAX_VL-1:128] ← 0
```

VFMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)

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

VFMADD132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] ← MAX_VL-1:128RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAX_VL-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[MAX_VL-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[MAX_VL-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);
VFMADD132SD __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);
VFMADD213SD __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);
VFMADD231SD __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 99 /r VFMADD132SS xmm0, xmm1, xmm2/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm2/m32, add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 A9 /r VFMADD213SS xmm0, xmm1, xmm2/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm1, add to xmm2/m32 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 B9 /r VFMADD231SS xmm0, xmm1, xmm2/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2/m32, add to xmm0 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W0 99 /r VFMADD132SS xmm0 {k1}{z}, xmm1, xmm2/m32{er}	F1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm0 and xmm2/m32, add to xmm1 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W0 A9 /r VFMADD213SS xmm0 {k1}{z}, xmm1, xmm2/m32{er}	F1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm0 and xmm1, add to xmm2/m32 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W0 B9 /r VFMADD231SS xmm0 {k1}{z}, xmm1, xmm2/m32{er}	F1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2/m32, add to xmm0 and put result in xmm0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
F1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

VFMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding

and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

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

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

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

Operation

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

VFMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

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

VFMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

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

VFMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0]] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-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[MAX_VL-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[MAX_VL-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[MAX_VL-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);
VFMADD132SS __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);
VFMADD213SS __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);
VFMADD231SS __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, add/subtract elements in xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, add/subtract elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add/subtract elements in xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, add/subtract elements in ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, add/subtract elements in ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, add/subtract elements in ymm0 and put result in ymm0.
EVEX.DDS.512.66.0F38.W1 A6 /r VFMADDSUB213PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm0 and zmm1, add/subtract elements in zmm2/m512/m64bcst and put result in zmm0 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 B6 /r VFMADDSUB231PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2/m512/m64bcst, add/subtract elements in zmm0 and put result in zmm0 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 96 /r VFMADDSUB132PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm0 and zmm2/m512/m64bcst, add/subtract elements in zmm1 and put result in zmm0 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

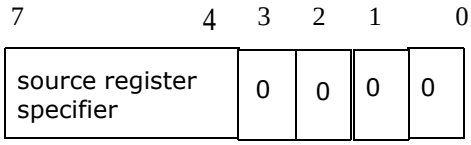
VFMADDSUB132PD: Multiplies the two, four, or eight packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point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

resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

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

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

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in



Bits[3:0]: Reserved for Future Use and must be Zero

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.

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[MAX_VL-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[MAX_VL-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[MAX_VL-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) = (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
                ELSE DEST[i+63:i] ←
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
```

VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (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
```

VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) =(8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                ELSE DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
```

```
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
```

VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (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
```

VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
```

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

VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (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
```

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);
VFMADDSUB132PD __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);
VFMADDSUB213PD __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);
VFMADDSUB231PD __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);
VFMADDSUB132PD __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);
VFMADDSUB213PD __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);
VFMADDSUB231PD __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, add/subtract elements in xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 A6 /r VFMADDSUB213PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, add/subtract elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 B6 /r VFMADDSUB231PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, add/subtract elements in xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, add/subtract elements in ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, add/subtract elements in ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, add/subtract elements in ymm0 and put result in ymm0.
EVEX.DDS.512.66.0F38.W0 A6 /r VFMADDSUB213PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm0 and zmm1, add/subtract elements in zmm2/m512/m32bcst and put result in zmm0 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 B6 /r VFMADDSUB231PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2/m512/m32bcst, add/subtract elements in zmm0 and put result in zmm0 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 96 /r VFMADDSUB132PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm0 and zmm2/m512/m32bcst, add/subtract elements in zmm1 and put result in zmm0 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

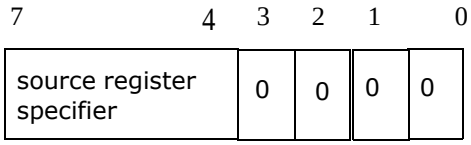
VFMADDSUB132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision 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) is a ZMM register and encoded in



Bits[3:0]: Reserved for Future Use and must be Zero

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.

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[MAX_VL-1:128] ←0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ←0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ←0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
```

```
                ELSE DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
```

VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (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
```

VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
```

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

VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (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
```

VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

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

```
(KL, VL) = (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
```

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);
VFMADDSUB132PS __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);
VFMADDSUB213PS __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);
VFMADDSUB231PS __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);
VFMADDSUB132PS __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);
VFMADDSUB213PS __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);
VFMADDSUB231PS __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, subtract/add elements in xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, subtract/add elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, subtract/add elements in xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, subtract/add elements in ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, subtract/add elements in ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, subtract/add elements in ymm0 and put result in ymm0.
EVEX.DDS.512.66.0F38.W1 97 /r VFMSUBADD132PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm0 and zmm2/m512/m64bcst, subtract/add elements in zmm1 and put result in zmm0 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 A7 /r VFMSUBADD213PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm0 and zmm1, subtract/add elements in zmm2/m512/m64bcst and put result in zmm0 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 B7 /r VFMSUBADD231PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2/m512/m64bcst, subtract/add elements in zmm0 and put result in zmm0 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

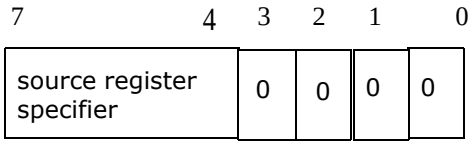
VFMSUBADD132PD: Multiplies the two, four, or eight packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision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

resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

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

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

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in



Bits[3:0]: Reserved for Future Use and must be Zero

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.

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[MAX_VL-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[MAX_VL-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[MAX_VL-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) = (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
                ELSE DEST[i+63:i] ←
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
```

VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (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
```

VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
                ELSE DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
```

```
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
```

VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (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
```

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

```
(KL, VL) = (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
```

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

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

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);
VFMSUBADD132PD __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);
VFMSUBADD213PD __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);
VFMSUBADD231PD __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);
VFMSUBADD132PD __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);
VFMSUBADD213PD __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);
VFMSUBADD231PD __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, subtract/add elements in xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 A7 /r VFMSUBADD213PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, subtract/add elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 B7 /r VFMSUBADD231PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, subtract/add elements in xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, subtract/add elements in ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, subtract/add elements in ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, subtract/add elements in ymm0 and put result in ymm0.
EVEX.DDS.512.66.0F38.W0 97 /r VFMSUBADD132PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm0 and zmm2/m512/m32bcst, subtract/add elements in zmm1 and put result in zmm0 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 A7 /r VFMSUBADD213PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm0 and zmm1, subtract/add elements in zmm2/m512/m32bcst and put result in zmm0 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 B7 /r VFMSUBADD231PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2/m512/m32bcst, subtract/add elements in zmm0 and put result in zmm0 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

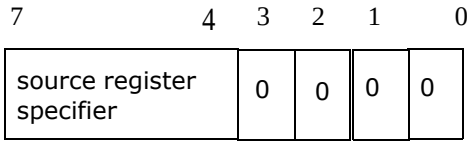
VFMSUBADD132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds

the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in



Bits[3:0]: Reserved for Future Use and must be Zero

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.

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[MAX_VL-1:128] ←0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ←0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ←0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
```

```
                ELSE DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
```

VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (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
```

VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
```

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

VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (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
```

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

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

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

```
(KL, VL) = (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
```

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);
VFMSUBADD132PS __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);
VFMSUBADD213PS __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);
VFMSUBADD231PS __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);
VFMSUBADD132PS __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);
VFMSUBADD213PS __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);
VFMSUBADD231PS __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W1 9A /r VFMSUB132PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.NDS.128.66.0F38.W1 AA /r VFMSUB213PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, subtract xmm2/mem and put result in xmm0.
VEX.NDS.128.66.0F38.W1 BA /r VFMSUB231PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.
VEX.NDS.256.66.0F38.W1 9A /r VFMSUB132PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, subtract ymm1 and put result in ymm0.
VEX.NDS.256.66.0F38.W1 AA /r VFMSUB213PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, subtract ymm2/mem and put result in ymm0.
VEX.NDS.256.66.0F38.W1 BA /r VFMSUB231PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, subtract ymm0 and put result in ymm0.
EVEX.NDS.512.66.0F38.W1 9A /r VFMSUB132PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm0 and zmm2/m512/m64bcst, subtract to zmm1 and put result in zmm0 subject to writemask k1.
EVEX.NDS.512.66.0F38.W1 AA /r VFMSUB213PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm0 and zmm1, subtract to zmm2/m512/m64bcst and put result in zmm0 subject to writemask k1.
EVEX.NDS.512.66.0F38.W1 BA /r VFMSUB231PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2/m512/m64bcst, subtract to zmm0 and put result in zmm0 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

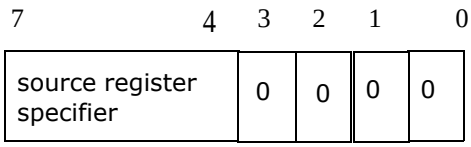
Performs a set of SIMD multiply-subtract computation on packed double-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register.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

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

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

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

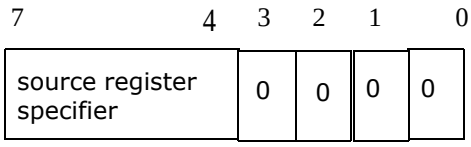
EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in



Bits[3:0]: Reserved for Future Use and must be Zero

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



Bits[3:0]: Reserved for Future Use and must be Zero

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 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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

VFMSUB213PD 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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

VFMSUB231PD 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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
```

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

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

```
(KL, VL) = (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
```

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

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

```
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
```

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

```
(KL, VL) = (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
```

VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (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
```

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);
VFMSUB132PD __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);
VFMSUB213PD __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);
VFMSUB231PD __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);
VFMSUB132PD __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);
VFMSUB213PD __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);
VFMSUB231PD __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

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

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 9A /r VFMSUB132PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.NDS.128.66.0F38.W0 AA /r VFMSUB213PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, subtract xmm2/mem and put result in xmm0.
VEX.NDS.128.66.0F38.W0 BA /r VFMSUB231PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.
VEX.NDS.256.66.0F38.W0 9A /r VFMSUB132PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, subtract ymm1 and put result in ymm0.
VEX.NDS.256.66.0F38.W0 AA /r VFMSUB213PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, subtract ymm2/mem and put result in ymm0.
VEX.NDS.256.66.0F38.0 BA /r VFMSUB231PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, subtract ymm0 and put result in ymm0.
EVEX.NDS.512.66.0F38.W0 9A /r VFMSUB132PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm0 and zmm2/m512/m32bcst, subtract to zmm1 and put result in zmm0.
EVEX.NDS.512.66.0F38.W0 AA /r VFMSUB213PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm0 and zmm1, subtract to zmm2/m512/m32bcst and put result in zmm0.
EVEX.NDS.512.66.0F38.W0 BA /r VFMSUB231PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2/m512/m32bcst, subtract to zmm0 and put result in zmm0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

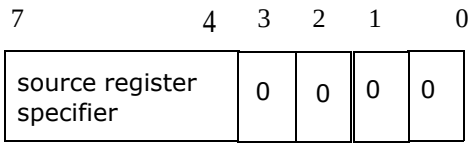
Performs a set of SIMD multiply-subtract computation on packed single-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in



Bits[3:0]: Reserved for Future Use and must be Zero

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 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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
```

```
    DEST[MAX_VL-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

```
    FI;
ENDFOR
```

VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (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
```

VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (16, 512)

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

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

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

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

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

```
(KL, VL) = (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
```

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);
VFMSUB132PS __m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);
VFMSUB213PS __m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);
VFMSUB231PS __m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);
VFMSUB132PS __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);
VFMSUB213PS __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);
VFMSUB231PS __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9B /r VFMSUB132SD xmm0, xmm1, xmm2/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm2/m64, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 AB /r VFMSUB213SD xmm0, xmm1, xmm2/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm1, subtract xmm2/m64 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 BB /r VFMSUB231SD xmm0, xmm1, xmm2/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2/m64, subtract xmm0 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W1 9B /r VFMSUB132SD xmm0 {k1}{z}, xmm1, xmm2/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm0 and xmm2/m64, subtract xmm1 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W1 AB /r VFMSUB213SD xmm0 {k1}{z}, xmm1, xmm2/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm0 and xmm1, subtract xmm2/m64 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W1 BB /r VFMSUB231SD xmm0 {k1}{z}, xmm1, xmm2/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2/m64, subtract xmm0 and put result in xmm0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

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

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

VFMSUB231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the first source operand, performs

rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

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

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

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

Operation

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

VFMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

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

VFMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

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

VFMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundFPControl(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-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[MAX_VL-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[MAX_VL-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[MAX_VL-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);
VFMSUB132SD __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);
VFMSUB213SD __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);
VFMSUB231SD __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9B /r VFMSUB132SS xmm0, xmm1, xmm2/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm2/m32, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 AB /r VFMSUB213SS xmm0, xmm1, xmm2/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm1, subtract xmm2/m32 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 BB /r VFMSUB231SS xmm0, xmm1, xmm2/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2/m32, subtract xmm0 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W0 9B /r VFMSUB132SS xmm0 {k1}{z}, xmm1, xmm2/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm0 and xmm2/m32, subtract xmm1 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W0 AB /r VFMSUB213SS xmm0 {k1}{z}, xmm1, xmm2/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm0 and xmm1, subtract xmm2/m32 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W0 BB /r VFMSUB231SS xmm0 {k1}{z}, xmm1, xmm2/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2/m32, subtract xmm0 and put result in xmm0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD multiply-subtract computation on the low packed single-precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 32-bit memory location.

VFMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

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

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

and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

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

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

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

Operation

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

VFMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)

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

VFMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)

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

VFMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[63:0] - DEST[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-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[MAX_VL-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[MAX_VL-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[MAX_VL-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);
VFMSUB132SS __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);
VFMSUB213SS __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);
VFMSUB231SS __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.

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.NDS.128.66.0F38.W1 9C /r VFNMADD132PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and add to xmm1 and put result in xmm0.
VEX.NDS.128.66.0F38.W1 AC /r VFNMADD213PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, negate the multiplication result and add to xmm2/mem and put result in xmm0.
VEX.NDS.128.66.0F38.W1 BC /r VFNMADD231PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and add to xmm0 and put result in xmm0.
VEX.NDS.256.66.0F38.W1 9C /r VFNMADD132PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and add to ymm1 and put result in ymm0.
VEX.NDS.256.66.0F38.W1 AC /r VFNMADD213PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, negate the multiplication result and add to ymm2/mem and put result in ymm0.
VEX.NDS.256.66.0F38.W1 BC /r VFNMADD231PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and add to ymm0 and put result in ymm0.
EVEX.NDS.512.66.0F38.W1 9C /r VFNMADD132PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm0 and zmm2/m512/m64bcst, negate the multiplication result and add to zmm1 and put result in zmm0.
EVEX.NDS.512.66.0F38.W1 AC /r VFNMADD213PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm0 and zmm1, negate the multiplication result and add to zmm2/m512/m64bcst and put result in zmm0.
EVEX.NDS.512.66.0F38.W1 BC /r VFNMADD231PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2/m512/m64bcst, negate the multiplication result and add to zmm0 and put result in zmm0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

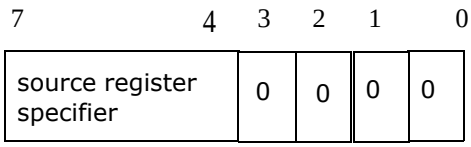
VFNMADD132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two, four or 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) is a ZMM register and encoded in



Bits[3:0]: Reserved for Future Use and must be Zero

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 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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (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
```

VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (8, 512)

```
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
```

VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (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
```

VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (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
```

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);
VFNMADD132PD __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);
VFNMADD213PD __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);
VFNMADD231PD __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);
VFNMADD132PD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);
VFNMADD213PD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);
VFNMADD231PD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

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.NDS.128.66.0F38.W0 9C /r VFNMADD132PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and add to xmm1 and put result in xmm0.
VEX.NDS.128.66.0F38.W0 AC /r VFNMADD213PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, negate the multiplication result and add to xmm2/mem and put result in xmm0.
VEX.NDS.128.66.0F38.W0 BC /r VFNMADD231PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and add to xmm0 and put result in xmm0.
VEX.NDS.256.66.0F38.W0 9C /r VFNMADD132PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and add to ymm1 and put result in ymm0.
VEX.NDS.256.66.0F38.W0 AC /r VFNMADD213PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, negate the multiplication result and add to ymm2/mem and put result in ymm0.
VEX.NDS.256.66.0F38.0 BC /r VFNMADD231PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and add to ymm0 and put result in ymm0.
EVEX.NDS.512.66.0F38.W0 9C /r VFNMADD132PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm0 and zmm2/m512/m32bcst, negate the multiplication result and add to zmm1 and put result in zmm0.
EVEX.NDS.512.66.0F38.W0 AC /r VFNMADD213PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm0 and zmm1, negate the multiplication result and add to zmm2/m512/m32bcst and put result in zmm0.
EVEX.NDS.512.66.0F38.W0 BC /r VFNMADD231PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2/m512/m32bcst, negate the multiplication result and add to zmm0 and put result in zmm0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

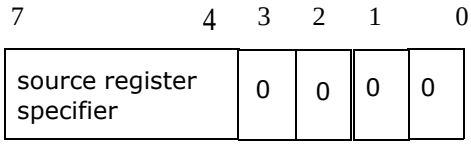
VFNMADD132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-preci-

sion 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).

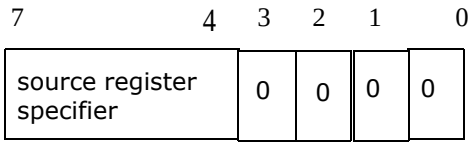
EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in



Bits[3:0]: Reserved for Future Use and must be Zero

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



Bits[3:0]: Reserved for Future Use and must be Zero

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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

```
                RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
        ELSE
                IF *merging-masking*                    ; merging-masking
                        THEN *DEST[i+31:i] remains unchanged*
                        ELSE                            ; zeroing-masking
                                DEST[i+31:i] ← 0
                FI
    FI;
ENDFOR
```

VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (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
```

VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

```
    FI;
ENDFOR
```

VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (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
```

VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (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
```

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);
VFNMADD132PS __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);
VFNMADD213PS __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);
VFNMADD231PS __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);
VFNMADD132PS __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);
VFNMADD213PS __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);
VFNMADD231PS __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

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.DDS.128.66.0F38.W1 9D /r VFNMADD132SD xmm0, xmm1, xmm2/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm2/mem, negate the multiplication result and add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 AD /r VFNMADD213SD xmm0, xmm1, xmm2/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm1, negate the multiplication result and add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 BD /r VFNMADD231SD xmm0, xmm1, xmm2/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2/mem, negate the multiplication result and add to xmm0 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W1 9D /r VFNMADD132SD xmm0 {k1}{z}, xmm1, xmm2/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm0 and xmm2/m64, negate the multiplication result and add to xmm1 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W1 AD /r VFNMADD213SD xmm0 {k1}{z}, xmm1, xmm2/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm0 and xmm1, negate the multiplication result and add to xmm2/m64 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W1 BD /r VFNMADD231SD xmm0 {k1}{z}, xmm1, xmm2/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2/m64, negate the multiplication result and add to xmm0 and put result in xmm0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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

VFNMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

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

VFNMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)

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

```
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-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[MAX_VL-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[MAX_VL-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[MAX_VL-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);
VFNMADD132SD __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);
VFNMADD213SD __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);
VFNMADD231SD __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 Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

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.DDS.LIG.128.66.0F38.W0 9D /r VFNMADD132SS xmm0, xmm1, xmm2/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm2/m32, negate the multiplication result and add to xmm1 and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W0 AD /r VFNMADD213SS xmm0, xmm1, xmm2/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm1, negate the multiplication result and add to xmm2/m32 and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W0 BD /r VFNMADD231SS xmm0, xmm1, xmm2/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2/m32, negate the multiplication result and add to xmm0 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W0 9D /r VFNMADD132SS xmm0 {k1}{z}, xmm1, xmm2/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm0 and xmm2/m32, negate the multiplication result and add to xmm1 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W0 AD /r VFNMADD213SS xmm0 {k1}{z}, xmm1, xmm2/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm0 and xmm1, negate the multiplication result and add to xmm2/m32 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W0 BD /r VFNMADD231SS xmm0 {k1}{z}, xmm1, xmm2/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2/m32, negate the multiplication result and add to xmm0 and put result in xmm0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(-(DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0
```

VFNMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

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

VFNMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)

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

```
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) + DEST[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-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[MAX_VL-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[MAX_VL-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[MAX_VL-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);
VFNMADD132SS __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);
VFNMADD213SS __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);
VFNMADD231SS __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 Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

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.NDS.128.66.0F38.W1 9E /r VFNMSUB132PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1 and put result in xmm0.
VEX.NDS.128.66.0F38.W1 AE /r VFNMSUB213PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, negate the multiplication result and subtract xmm2/mem and put result in xmm0.
VEX.NDS.128.66.0F38.W1 BE /r VFNMSUB231PD xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0 and put result in xmm0.
VEX.NDS.256.66.0F38.W1 9E /r VFNMSUB132PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and subtract ymm1 and put result in ymm0.
VEX.NDS.256.66.0F38.W1 AE /r VFNMSUB213PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, negate the multiplication result and subtract ymm2/mem and put result in ymm0.
VEX.NDS.256.66.0F38.W1 BE /r VFNMSUB231PD ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and subtract ymm0 and put result in ymm0.
EVEX.NDS.512.66.0F38.W1 9E /r VFNMSUB132PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm0 and zmm2/m512/m64bcst, negate the multiplication result and subtract to zmm1 and put result in zmm0.
EVEX.NDS.512.66.0F38.W1 AE /r VFNMSUB213PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm0 and zmm1, negate the multiplication result and subtract to zmm2/m512/m64bcst and put result in zmm0.
EVEX.NDS.512.66.0F38.W1 BE /r VFNMSUB231PD zmm0 {k1}{z}, zmm1, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2/m512/m64bcst, negate the multiplication result and subtract to zmm0 and put result in zmm0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

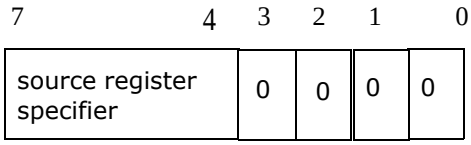
VFNMSUB132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision

floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

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

VFNMSUB231PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in



Bits[3:0]: Reserved for Future Use and must be Zero

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 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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (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
```

VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (8, 512)

```
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
```

VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (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
```

VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

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

```
(KL, VL) = (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
```

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);
VFNMSUB132PD __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);
VFNMSUB213PD __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);
VFNMSUB231PD __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);
VFNMSUB132PD __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);
VFNMSUB213PD __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);
VFNMSUB231PD __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

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.NDS.128.66.0F38.W0 9E /r VFNMSUB132PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1 and put result in xmm0.
VEX.NDS.128.66.0F38.W0 AE /r VFNMSUB213PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, negate the multiplication result and subtract xmm2/mem and put result in xmm0.
VEX.NDS.128.66.0F38.W0 BE /r VFNMSUB231PS xmm0, xmm1, xmm2/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0 and put result in xmm0.
VEX.NDS.256.66.0F38.W0 9E /r VFNMSUB132PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and subtract ymm1 and put result in ymm0.
VEX.NDS.256.66.0F38.W0 AE /r VFNMSUB213PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, negate the multiplication result and subtract ymm2/mem and put result in ymm0.
VEX.NDS.256.66.0F38.0 BE /r VFNMSUB231PS ymm0, ymm1, ymm2/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and subtract ymm0 and put result in ymm0.
EVEX.NDS.512.66.0F38.W0 9E /r VFNMSUB132PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm0 and s zmm2/m512/m32bcst, negate the multiplication result and subtract to zmm1 and put result in zmm0.
EVEX.NDS.512.66.0F38.W0 AE /r VFNMSUB213PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm0 and zmm1, negate the multiplication result and subtract to zmm2/m512/m32bcst and put result in zmm0.
EVEX.NDS.512.66.0F38.W0 BE /r VFNMSUB231PS zmm0 {k1}{z}, zmm1, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2/m512/m32bcst, negate the multiplication result subtract add to zmm0 and put result in zmm0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

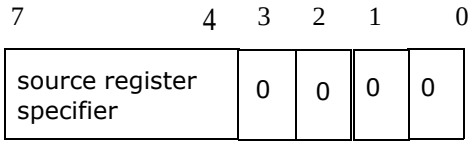
VFNMSUB132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-

precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

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

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

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in



Bits[3:0]: Reserved for Future Use and must be Zero

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 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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

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

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

VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (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
```

VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (16, 512)

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

VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (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
```

VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

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

```
(KL, VL) = (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
```

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);
VFNMSUB132PS __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);
VFNMSUB213PS __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);
VFNMSUB231PS __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);
VFNMSUB132PS __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);
VFNMSUB213PS __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);
VFNMSUB231PS __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

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.DDS.128.66.0F38.W1 9F /r VFNMSUB132SD xmm0, xmm1, xmm2/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 AF /r VFNMSUB213SD xmm0, xmm1, xmm2/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm1, negate the multiplication result and subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 BF /r VFNMSUB231SD xmm0, xmm1, xmm2/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W1 9F /r VFNMSUB132SD xmm0 {k1}{z}, xmm1, xmm2/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm0 and xmm2/m64, negate the multiplication result and subtract xmm1 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W1 AF /r VFNMSUB213SD xmm0 {k1}{z}, xmm1, xmm2/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm0 and xmm1, negate the multiplication result and subtract xmm2/m64 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W1 BF /r VFNMSUB231SD xmm0 {k1}{z}, xmm1, xmm2/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2/m64, negate the multiplication result and subtract xmm0 and put result in xmm0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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

VFNMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

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

VFNMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)

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

```
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-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[MAX_VL-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[MAX_VL-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[MAX_VL-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);
VFNMSUB132SD __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);
VFNMSUB213SD __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);
VFNMSUB231SD __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

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.DDS.128.66.0F38.W0 9F /r VFNMSUB132SS xmm0, xmm1, xmm2/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm2/m32, negate the multiplication result and subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 AF /r VFNMSUB213SS xmm0, xmm1, xmm2/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm1, negate the multiplication result and subtract xmm2/m32 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 BF /r VFNMSUB231SS xmm0, xmm1, xmm2/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2/m32, negate the multiplication result and subtract xmm0 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W0 9F /r VFNMSUB132SS xmm0 {k1}{z}, xmm1, xmm2/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm0 and xmm2/m32, negate the multiplication result and subtract xmm1 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W0 AF /r VFNMSUB213SS xmm0 {k1}{z}, xmm1, xmm2/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm0 and xmm1, negate the multiplication result and subtract xmm2/m32 and put result in xmm0.
EVEX.DDS.LIG.66.0F38.W0 BF /r VFNMSUB231SS xmm0 {k1}{z}, xmm1, xmm2/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2/m32, negate the multiplication result and subtract xmm0 and put result in xmm0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(-(DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0
```

VFNMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)

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

VFNMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
```

```
FI;
IF k1[0] or *no writemask*
    THEN      DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) - DEST[31:0])
    ELSE
          IF *merging-masking*                    ; merging-masking
               THEN *DEST[31:0] remains unchanged*
               ELSE                               ; zeroing-masking
                    THEN DEST[31:0] ← 0
          FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-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[MAX_VL-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[MAX_VL-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[MAX_VL-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);
VFNMSUB132SS __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);
VFNMSUB213SS __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);
VFNMSUB231SS __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 92 /vsib VGATHERDPS zmm1 {k1}, vm32z	T1S	V/V	AVX512F	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.512.66.0F38.W1 92 /vsib VGATHERDPD zmm1 {k1}, vm32y	T1S	V/V	AVX512F	Using signed dword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (r, w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

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.

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 before up-conversion.

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.

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, 2 or 4 byte displacement

VGATHERDPS (EVEX encoded version)

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        THEN DEST[i+31:i] ←
            EM[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
```

VGATHERDPD (EVEX encoded version)

```
(KL, VL) = (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
```

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);
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);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12.

VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 93 /vsib VGATHERQPS ymm1 {k1}, vm64z	T1S	V/V	AVX512F	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.512.66.0F38.W1 93 /vsib VGATHERQPD zmm1 {k1}, vm64z	T1S	V/V	AVX512F	Using signed qword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (r, w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

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.

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 before up-conversion.

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.

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, 2 or 4 byte displacement

VGATHERQPS (EVEX encoded version)

```
(KL, VL) = (8, 256)
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[MAX_VL-1:VL/2] ← 0
```

VGATHERQPD (EVEX encoded version)

```
(KL, VL) = (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
```

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);
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);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12.

VGETEXPPD—Convert Exponents of Packed DP FP Values to DP FP Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 42 /r VGETEXPPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}	FV	V/V	AVX512F	Convert each biased exponent (bits 62:52) of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Extracts the biased exponents from the normalized DP FP representation of each qword data element of the source operand (the second operand) as unbiased signed integer value. Each integer value of the unbiased exponent is converted to double-precision FP value and written to the corresponding qword elements of the destination operand (the first operand) as DP FP numbers.

The destination operand is a ZMM register and updated under the writemask. 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.

EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a FP number.

The formula is:

$GETEXP(x) = floor(\log_2(|x|))$

Notation **floor(x)** stands for the greatest integer not exceeding real number x.

Operation

```
NormalizeExpTinyDPFP(SRC[63:0])
{
    ; Jbit is the hidden integral bit of a FP number. In case of denormal number it has the value of ZERO.
    Src.Jbit ← 0;
    Dst.exp ← 1;
    Dst.fraction ← SRC[51:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit ← Dst.fraction[51];            ; Get the fraction MSB
        Dst.fraction ← Dst.fraction << 1 ;      ; One bit shift left
        Dst.exp-- ;                             ; Decrement the exponent
    }
    Dst.fraction ← 0;                           ; zero out fraction bits
    Dst.sign ← 1;                               ; Return negative sign
    TMP[63:0] ← MXCSR.DAZ? 0 : (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction) ;
    RETURN (TMP[63:0]);
}

ConvertExpDPFP(SRC[63:0])
{
    Src.sign ← 0;                               ; Zero out sign bit
    Src.exp ← SRC[62:52];
    Src.fraction ← SRC[51:0];
```

```
    ; Check for NaN
    IF (SRC = NaN) THEN
    {
        IF SRC = SNAN THEN SET IE;
        Return QNAN(SRC);
    }

    ; Check for +INF
    IF (SRC = +INF) THEN
    {
        Return (SRC);
    }

    ; check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) THEN
    {
        RETURN (-INF)
    }
    ELSE        ; check if denormal operand (notice that MXCSR.DAZ = 0)
    IF ((Src.exp = 0) AND (Src.fraction != 0)) THEN
    {
        TMP[63:0] ← NormalizeExpTinyDPFP(SRC[63:0]) ;        ; Get Normalized Exponent
        Set #DE
    }
    ELSE        ; exponent value is correct
    {
        Dst.fraction ← 0;        ; zero out fraction bits
        TMP[63:0] ← (Src.sign << 63) OR (Src.exp << 52) OR (Src.fraction) ;
    }
    TMP ← SAR(TMP, 52) ;        ; Shift Arithmetic Right
    TMP ← TMP – 1023;        ; Subtract Bias
    RETURN CvtI2D(TMP) ;        ; Convert INT to Double-Precision FP number
}
```

VGETEXPPD (EVEX encoded versions)

```
(KL, VL) = (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
```

Illustrative examples can be found under the VGETEXPPS instruction.

Table 5-11. VGETEXPPD Special Cases

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	No Exceptions
0 < |src1| < INF	floor($\log_2$(|src1|))	
| src1| = +INF	+INF	
| src1| = 0	-INF	

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);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E2.

VGETEXPPS—Convert Exponents of Packed SP FP Values to SP FP Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 42 /r VGETEXPPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}	FV	V/V	AVX512F	Convert each biased exponent (bits 30:23) of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Extracts the biased exponents from the normalized SP FP representation of each dword data element of the source operand (the second operand) as unbiased signed integer value. Each integer value of the unbiased exponent is converted to single-precision FP value and written to the corresponding dword elements of the destination operand (the first operand) as SP FP numbers.

The destination operand is a ZMM register and updated under the writemask. 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.

EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a FP number.

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 FP exceptions.

Operation

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

ConvertExpSPFP(SRC[31:0])
{
    Src.sign ← 0;                             ; Zero out sign bit
```

```
    Src.exp ← SRC[30:23];
    Src.fraction ← SRC[22:0];

    ; Check for NaN
    IF (SRC = NaN) THEN
    {
        IF SRC = SNAN THEN SET IE;
        Return QNAN(SRC);
    }

    ; Check for +INF
    IF (SRC = +INF) THEN
    {
        Return (SRC);
    }

    ; check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) THEN
    {
        RETURN (-INF)
    }
    ELSE            ; check if denormal operand (notice that MXCSR.DAZ = 0)
    IF ((Src.exp = 0) AND (Src.fraction != 0)) THEN
    {
        TMP[31:0] ← NormalizeExpTinySPFP(SRC[31:0]) ;          ; Get Normalized Exponent
        Set #DE
    }
    ELSE            ; exponent value is correct
    {
        Dst.fraction ← 0;               ; zero out fraction bits
        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 Cvtl2S(TMP) ;            ; Convert INT to Single-Precision FP number
}
```

VGETEXPPS (EVEX encoded versions)

```
(KL, VL) = (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
```

The below figure illustrates the VGETEXPPS functionality.

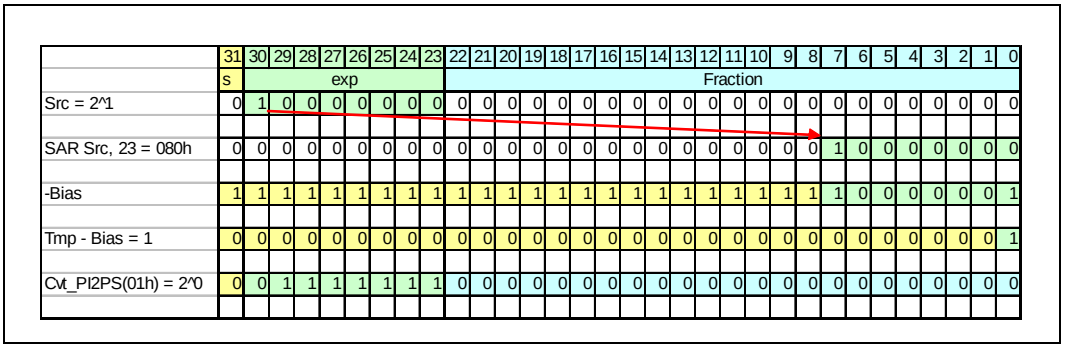


Figure 5-22. VGETEXPPS Functionality

Table 5-12. VGETEXPPS Special Cases

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	No Exceptions
0 < |src1| < INF	floor($\log_2$(|src1|))	
| src1| = +INF	+INF	
| src1| = 0	-INF	

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);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E2.

VGETEXPSD—Convert Exponents of Scalar DP FP Values to DP FP Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 43 /r VGETEXPSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	T1S	V/V	AVX512F	Convert a biased exponent (bits 62:52) of packed double-precision floating-point value in the source operand to DP FP results representing unbiased integer exponent and stores the result in the destination register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Extracts the biased exponent from the normalized DP FP representation of the low qword data element of the source operand (the third operand) as unbiased signed integer value. The integer value of the unbiased exponent is converted to double-precision FP value and written to the destination operand (the first operand) as DP FP numbers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float64 memory location. The low quadword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number.

The formula is:

GETEXP(x) = floor($\log_2(|x|)$)

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Operation

```
NormalizeExpTinyDPFP(SRC[63:0])
{
    ; Jbit is the hidden integral bit of a FP number. In case of denormal number it has the value of ZERO.
    Src.Jbit ← 0;
    Dst.exp ← 1;
    Dst.fraction ← SRC[51:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit ← Dst.fraction[51];          ; Get the fraction MSB
        Dst.fraction ← Dst.fraction << 1 ;    ; One bit shift left
        Dst.exp-- ;                           ; Decrement the exponent
    }
    Dst.fraction ← 0;                 ; zero out fraction bits
    Dst.sign ← 1;                     ; Return negative sign
    TMP[63:0] ← MXCSR.DAZ? 0 : (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction) ;
    RETURN (TMP[63:0]);
}

ConvertExpDPFP(SRC[63:0])
{
    Src.sign ← 0;                 ; Zero out sign bit
    Src.exp ← SRC[62:52];
    Src.fraction ← SRC[51:0];
```

```
    ; Check for NaN
    IF (SRC = NaN) THEN
    {
        IF SRC = SNAN THEN SET IE;
        Return QNAN(SRC);
    }

    ; Check for +INF
    IF (SRC = +INF) THEN
    {
        Return (SRC);
    }

    ; check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) THEN
    {
        RETURN (-INF)
    }
    ELSE        ; check if denormal operand (notice that MXCSR.DAZ = 0)
    IF ((Src.exp = 0) AND (Src.fraction != 0)) THEN
    {
        TMP[63:0] ← NormalizeExpTinyDPFP(SRC[63:0]) ;        ; Get Normalized Exponent
        IF (MXCSR.DAZ = 0) Set #DE
}
    ELSE        ; exponent value is correct
    {
        Dst.fraction ← 0;       ; zero out fraction bits
        TMP[63:0] ← (Src.sign << 63) OR (Src.exp << 52) OR (Src.fraction) ;
    }
    TMP ← SAR(TMP, 52) ;        ; Shift Arithmetic Right
    TMP ← TMP – 1023;           ; Subtract Bias
    RETURN CvtI2D(TMP) ;        ; Convert INT to Double-Precision FP number
}
```

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[MAX_VL-1:128] ← 0
```

Illustrative examples can be found under the VGETEXPPS instruction.

Table 5-13. VGETEXPSD Special Cases

Input Operand	Result	Comments
src2 = NaN	QNaN(src2)	No Exceptions
0 < |src2| < INF	floor($\log_2$(|src2|))	
| src2| = +INF	+INF	
| src2| = 0	-INF	

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E3.

VGETEXPSS—Convert Exponents of Scalar SP FP Values to SP FP Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 43 /r VGETEXPSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	T1S	V/V	AVX512F	Convert a biased exponent (bits 30:23) of packed single-precision floating-point value in the source operand to SP FP result representing unbiased integer exponent and stores the result in the destination register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Extracts the biased exponent from the normalized SP FP representation of the low doubleword data element of the source operand (the third operand) as unbiased signed integer value. The integer value of the unbiased exponent is converted to single-precision FP value and written to the destination operand (the first operand) as SP FP numbers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float32 memory location. The the low doubleword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number.

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 FP exceptions.

Operation

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

ConvertExpSPFP(SRC[31:0])
{
    Src.sign ← 0;                             ; Zero out sign bit
```

```
    Src.exp ← SRC[30:23];
    Src.fraction ← SRC[22:0];

    ; Check for NaN
    IF (SRC = NaN) THEN
    {
        IF SRC = SNAN THEN SET IE;
        Return QNAN(SRC);
    }

    ; Check for +INF
    IF (SRC = +INF) THEN
    {
        Return (SRC);
    }

    ; check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) THEN
    {
        RETURN (-INF)
    }
    ELSE        ; check if denormal operand (notice that MXCSR.DAZ = 0)
    IF ((Src.exp = 0) AND (Src.fraction != 0)) THEN
    {
        TMP[31:0] ← NormalizeExpTinySPFP(SRC[31:0]) ;       ; Get Normalized Exponent
        Set #DE
    }
    ELSE        ; exponent value is correct
    {
        Dst.fraction ← 0;           ; zero out fraction bits
        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 Cvtl2S(TMP) ;            ; Convert INT to Single-Precision FP number
}
```

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[MAX_VL-1:128] ← 0
```

Table 5-14. VGETEXPSS Special Cases

Input Operand	Result	Comments
src2 = NaN	QNaN(src2)	No Exceptions
0 < |src2| < INF	floor($\log_2$(|src2|))	
| src2| = +INF	+INF	
| src2| = 0	-INF	

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPSS __m128 _mm_getexp_ss(__m128 a, __m128 b);
VGETEXPSS __m128 _mm_mask_getexp_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_maskz_getexp_ss(__mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_getexp_round_ss(__m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_mask_getexp_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_maskz_getexp_round_ss(__mmask8 k, __m128 a, __m128 b, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E3.

VGETMANTPD—Extract Float64 Vector of Normalized Mantissas from Float64 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F3A.W1 26 /r ib VGETMANTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	FV	V/V	AVX512F	Get Normalized Mantissa from float64 vector zmm2/m512/m64bcst and store the result in zmm1, using *imm*8 for sign control and mantissa interval normalization, under writemask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FVI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA

Description

Convert double-precision floating values in the source operand (the second operand) to DP FP values with the mantissa normalization and sign control specified by the imm8, 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 register updated under the writemask. 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.

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

$$GetMant(x) = \pm 2^{k}|x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then k = -1, otherwise K = 0. The encoded value of imm8[1:0] and sc are shown in Figure 5-23.

Each converted DP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows the table below when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

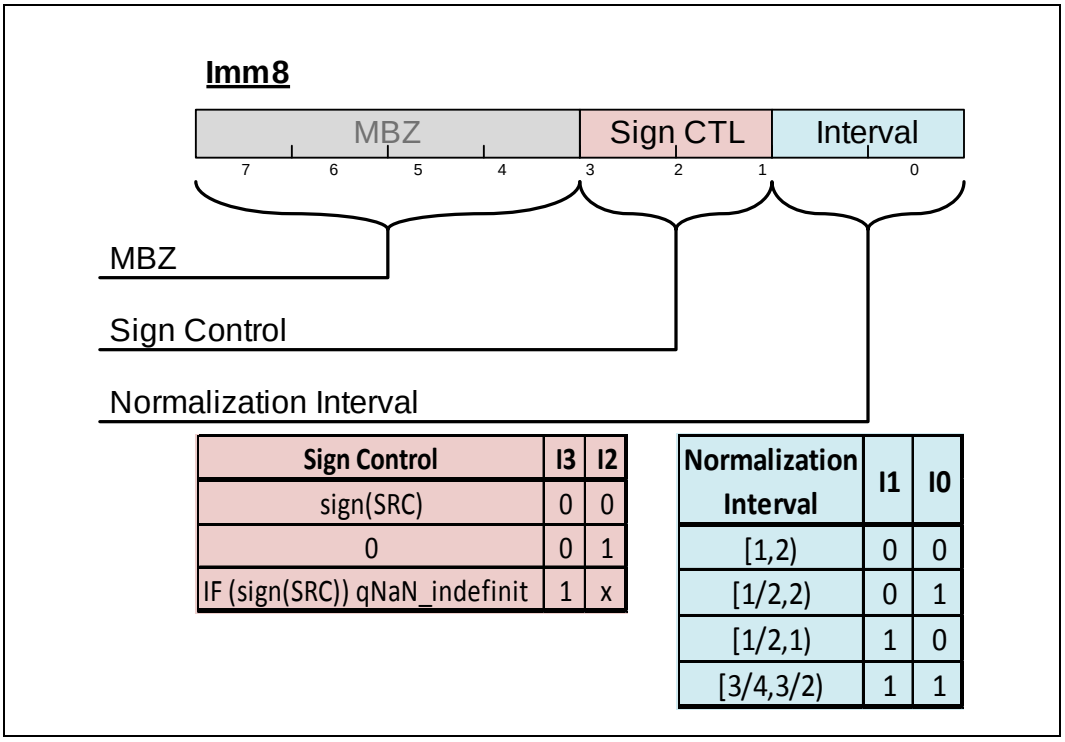
Note: EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Table 5-15. 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)	If (SC[1]) then #IE



Sign Control	I3	I2
sign(SRC)	0	0
0	0	1
IF (sign(SRC)) qNaN_indefinit	1	x

Normalization Interval	I1	I0
[1,2)	0	0
[1/2,2)	0	1
[1/2,1)	1	0
[3/4,3/2)	1	1

Figure 5-23. Graphic View of Imm8

Operation

```
GetNormalizeMantissa(SRC[63:0], SignCtrl[1:0], Interv[1:0])
{

    ; Extracting the SRC sign, exponent and mantissa fields
    Dst.sign ← SignCtrl[0] ? 0 : Src[63];                ; Get sign bit
Dst.exp ← SRC[62:52];; Get original exponent value
Dst.fraction ← SRC[51:0];; Get original fraction value
ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
```

```
DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
InfiniteOperand ← (Dst.exp = 07FFh) AND (Dst.fraction = 0);
NaNOperand ← (Dst.exp = 07FFh) AND (Dst.fraction != 0);

    ; Check for NAN operand
    IF (NaNOperand) THEN {
    IF (SRC = SNaN) THEN {Set #IE}
RETURN QNAN(SRC);
    }

    ; Check for Zero and Infinite operands
    IF ((ZeroOperand) OR (InfiniteOperand) THEN {
    Dst.exp ← 03FFh;              ; Override exponent with BIAS
RETURN ((Dst.sign<<63) | (Dst.exp<<52) | (Dst.fraction));
    }

    ; Check for negative operand (including -0.0)
    IF ((Src[63] = 1) AND SignCtrl[1]) THEN {
Set #IE;
RETURN QNaN_Indefinite;
    }

; Checking for denormal operands
IF (DenormOperand) THEN {
    IF (MXCSR.DAZ=1) {
Dst.fraction ← 0;                   ; Zero out fraction
    }
    ELSE {
        ; Jbit is the hidden integral bit. Zero in case of denormal operand.
Src.Jbit ← 0;                       ; Zero Src Jbit
        Dst.exp ← 03FFh;        ; Override exponent with BIAS
        WHILE (Src.Jbit = 0) {    ; normalize mantissa
            Src.Jbit ← Dst.fraction[51];        ; Get the fraction MSB
Dst.fraction ← (Dst.fraction << 1);         ; Start normalizing the mantissa
            Dst.exp-- ;     ; Adjust the exponent
    }
    SET #DE;            ; Set DE bit
}
}       ; At this point, Dst.fraction is normalized.

; Checking for exponent response
Unbiased.exp ← Dst.exp – 03FFh;         ; subtract the bias from exponent
IsOddExp ← Unbiased.exp[0];             ; recognized unbiased ODD exponent
SignalingBit ← Dst.fraction[51];

CASE (interv[1:0]) of
    00: Dst.exp ← 03FFh;            ; This is the bias
    01: Dst.exp ← (IsOddExp) ? 03FEh : 03FFh;           ; either bias-1, or bias
    10: Dst.exp ← 03FEh;            ; bias-1
    11: Dst.exp ← (SignalingBit) ? 03FEh : 03FFh;       ; either bias-1, or bias
; At this point Dst.exp has the correct result.

; Form the final destination
DEST[63:0] ← (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);
```

```
RETURN (DEST);
}

SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];
```

VGETMANTPD (EVEX encoded versions)

```
(KL, VL) = (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] ←   GetNormalizedMantissa(SRC[63:0], sc, interv)
                ELSE
                    DEST[i+63:i] ←   GetNormalizedMantissa(SRC[i+63:i], sc, interv)
            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
```

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);

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Exceptions Type E2.

VGETMANTPS—Extract Float32 Vector of Normalized Mantissas from Float32 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F3A.W0 26 /r ib VGETMANTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	FV	V/V	AVX512F	Get Normalized Mantissa from float32 vector zmm2/m512/m32bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FVI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA

Description

Convert single -precision floating values in the source operand (the second operand) to SP FP values with the mantissa normalization and sign control specified by the imm8, 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 register updated under the writemask. 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.

For each input SP FP value x, The conversion operation is:

$$GetMant(x) = \pm 2^{k}|x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then k = -1, otherwise K = 0. The encoded value of imm8[1:0] and sc are shown in Figure 5-23.

Each converted DP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows the table below when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

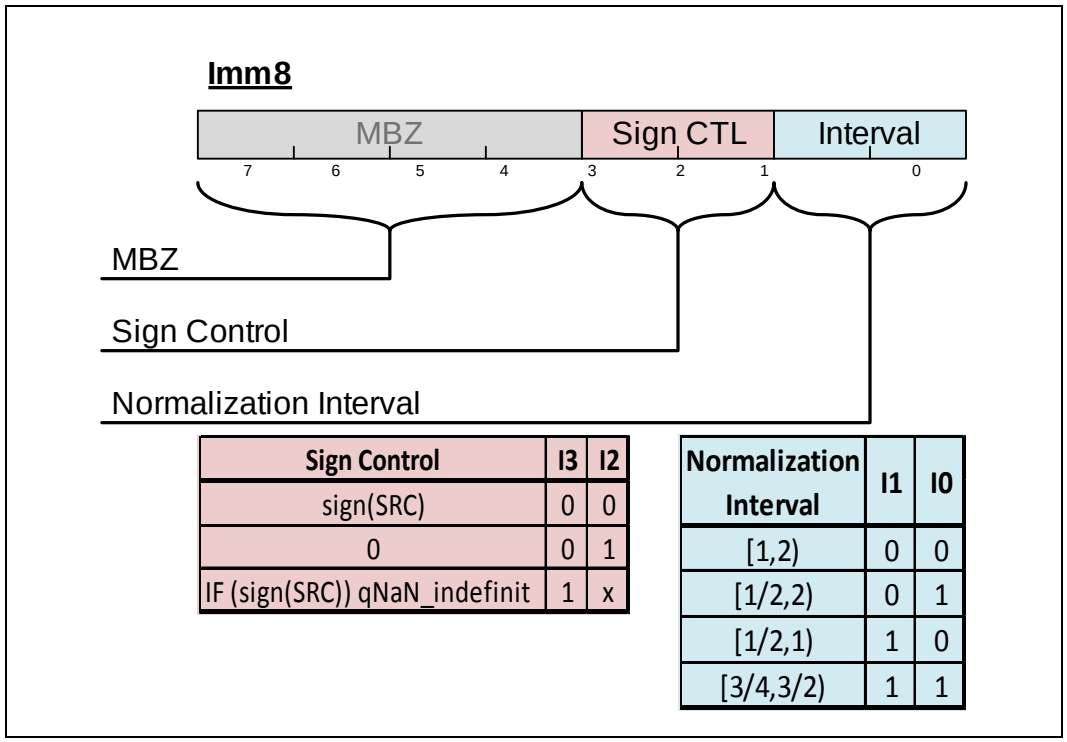
Note: EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Table 5-16. 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)	If (SC[1]) then #IE



Sign Control	I3	I2
sign(SRC)	0	0
0	0	1
IF (sign(SRC)) qNaN_indefinit	1	x

Normalization Interval	I1	I0
[1,2)	0	0
[1/2,2)	0	1
[1/2,1)	1	0
[3/4,3/2)	1	1

Figure 5-24. Graphic View of Imm8

Operation

```
GetNormalizeMantissa(SRC[31:0] , SignCtrl[1:0], Interv[1:0])
{
    ; Extracting the SRC sign, exponent and mantissa fields
    Dst.sign ← SignCtrl[0] ? 0 : Src[31];          ; Get sign bit
Dst.exp ← SRC[30:23];                  ; Get original exponent value
Dst.fraction ← SRC[22:0];              ; Get original fraction value
```

```
ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
InfiniteOperand ← (Dst.exp = 0FFh) AND (Dst.fraction = 0);
NaNOperand ← (Dst.exp = 0FFh) AND (Dst.fraction != 0);

    ; Check for NAN operand
    IF (NaNOperand) THEN {
    IF (SRC = SNaN) THEN {Set #IE}
RETURN QNAN(SRC);
    }

    ; Check for Zero and Infinite operands
    IF ((ZeroOperand) OR (InfiniteOperand) THEN
{
    Dst.exp ← 07Fh;              ; Override exponent with BIAS
RETURN ((Dst.sign<<31) | (Dst.exp<<23) | (Dst.fraction));
    }

    ; Check for negative operand (including -0.0)
    IF ((Src[31] = 1) AND SignCtrl[1]) THEN
{
Set #IE;
RETURN QNaN_Indefinite;
    }

; Checking for denormal operands
IF (DenormOperand) THEN {
    IF (MXCSR.DAZ=1) {
Dst.fraction ← 0;                    ; Zero out fraction
    }
    ELSE {
        ; Jbit is the hidden integral bit. Zero in case of denormal operand.
Src.Jbit ← 0;            ; Zero Src Jbit
        Dst.exp ← 07Fh;              ; Override exponent with BIAS
        WHILE (Src.Jbit = 0)         ; normalize mantissa
        {
            Src.Jbit ← Dst.fraction[22];      ; Get the fraction MSB
Dst.fraction ← (Dst.fraction << 1);       ; Start normalizing the mantissa
            Dst.exp-- ;          ; Adjust the exponent
    }
    SET #DE;             ; Set DE bit
}
}       ; At this point, Dst.fraction is normalized.

; Checking for exponent response
Unbiased.exp ← Dst.exp – 07Fh;               ; subtract the bias from exponent
IsOddExp ← Unbiased.exp[0];                  ; recognized unbiased ODD exponent
SignalingBit ← Dst.fraction[22];

CASE (interv[1:0]) of
    00: Dst.exp ← 07Fh;                  ; This is the bias
    01: Dst.exp ← (IsOddExp) ? 07Eh : 07Fh;      ; either bias-1, or bias
    10: Dst.exp ← 07Eh;              ; bias-1
    11: Dst.exp ← (SignalingBit) ? 07Eh : 07Fh;      ; either bias-1, or bias
```

```
; At this point Dst.exp has the correct result.

; Form the final destination
DEST[31:0] ← (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);

RETURN (DEST);
}

SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];
```

VGETMANTPS (EVEX **encoded versions)**

```
(KL, VL) = (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] ←    GetNormalizedMantissa(SRC[31:0], sc, interv)
                ELSE
                    DEST[i+31:i] ←    GetNormalizedMantissa(SRC[i+31:i], sc, interv)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
```

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);
```

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Exceptions Type E2.

VGETMANTSD—Extract Float64 of Normalized Mantissas from Float64 Scalar

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 27 /r ib VGETMANTSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	T1S	V/V	AVX512F	Get Normalized Mantissa from float64 in xmm3/m64 and store the result in xmm1, using *imm*8 for sign control and mantissa interval normalization, under writemask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Convert the double-precision floating values in the low quadword element of the second source operand (the third operand) to DP FP value with the mantissa normalization and sign control specified by the imm8, 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 depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then k = -1, otherwise K = 0. The encoded value of imm8[1:0] and sc are shown in Figure 5-23.

The converted DP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows the table below when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

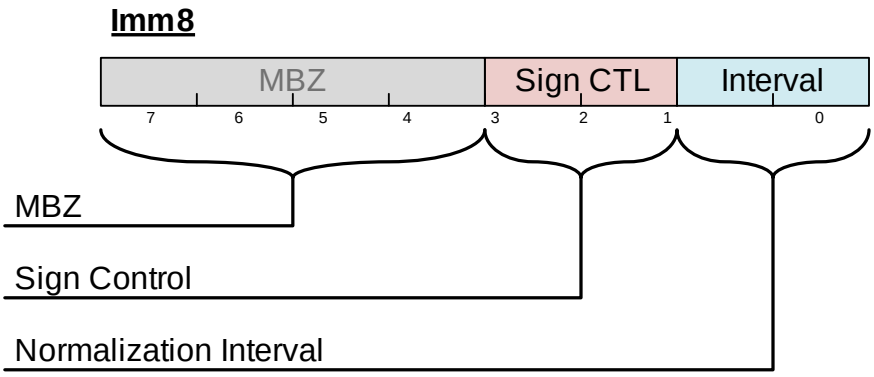
GetMant() special float values behavior

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)	If (SC[1]) then #IE

Below is a graphic view of Imm8



Sign Control	I3	I2
sign(SRC)	0	0
0	0	1
IF (sign(SRC)) qNaN_indefinit	1	x

Normalization Interval	I1	I0
[1,2)	0	0
[1/2,2)	0	1
[1/2,1)	1	0
[3/4,3/2)	1	1

Operation

```
GetNormalizeMantissa(SRC[63:0], SignCtrl[1:0], Interv[1:0])
{

    ; Extracting the SRC sign, exponent and mantissa fields
    Dst.sign ← SignCtrl[0] ? 0 : Src[63];          ; Get sign bit
Dst.exp ← SRC[62:52];                              ; Get original exponent value
Dst.fraction ← SRC[51:0];                          ; Get original fraction value
ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
InfiniteOperand ← (Dst.exp = 07FFh) AND (Dst.fraction = 0);
NaNOperand ← (Dst.exp = 07FFh) AND (Dst.fraction != 0);

    ; Check for NAN operand
    IF (NaNOperand) THEN {
    IF (SRC = SNaN) THEN {Set #IE}
RETURN QNAN(SRC);
    }

    ; Check for Zero and Infinite operands
    IF ((ZeroOperand) OR (InfiniteOperand) THEN {
    Dst.exp ← 03FFh;                   ; Override exponent with BIAS
RETURN ((Dst.sign<<63) | (Dst.exp<<52) | (Dst.fraction));
    }

    ; Check for negative operand (including -0.0)
    IF ((Src[63] = 1) AND SignCtrl[1]) THEN {
Set #IE;
RETURN QNaN_Indefinite;
    }
```

```
; Checking for denormal operands
IF (DenormOperand) THEN {
    IF (MXCSR.DAZ=1) {
Dst.fraction ← 0;                       ; Zero out fraction
    }
    ELSE {
        ; Jbit is the hidden integral bit. Zero in case of denormal operand.
Src.Jbit ← 0;                           ; Zero Src Jbit
        Dst.exp ← 03FFh;                ; Override exponent with BIAS
        WHILE (Src.Jbit = 0) {          ; normalize mantissa
            Src.Jbit ← Dst.fraction[51];        ; Get the fraction MSB
Dst.fraction ← (Dst.fraction << 1);             ; Start normalizing the mantissa
            Dst.exp-- ;                 ; Adjust the exponent
    }
    SET #DE;                    ; Set DE bit
}
}       ; At this point Dst.fraction is normalized.

; Checking for exponent response
Unbiased.exp ← Dst.exp – 03FFh;             ; subtract the bias from exponent
IsOddExp ← Unbiased.exp[0];                 ; recognized unbiased ODD exponent
SignalingBit ← Dst.fraction[51];

CASE (interv[1:0]) of
    00: Dst.exp ← 03FFh;                ; This is the bias
    01: Dst.exp ← (IsOddExp) ? 03FEh : 03FFh;       ; either bias-1, or bias
    10: Dst.exp ← 03FEh;                ; bias-1
    11: Dst.exp ← (SignalingBit) ? 03FEh : 03FFh;       ; either bias-1, or bias
; At this point Dst.exp has the correct result.

; Form the final destination
DEST[63:0] ← (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);

RETURN (DEST);
}

SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];
```

VGETMANTSD (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[63:0] ←
            GetNormalizedMantissa(SRC2[63:0], sc, interv)
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] ← 0
        FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTSD __m128d _mm_getmant_sd(__m128d a, __m128 b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_mask_getmant_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_maskz_getmant_sd(__mmask8 k, __m128 a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_getmant_round_sd(__m128d a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_mask_getmant_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_maskz_getmant_round_sd(__mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Exceptions Type E3.

VGETMANTSS—Extract Float32 Vector of Normalized Mantissas from Float32 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 27 /r ib VGETMANTSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	T1S	V/V	AVX512F	Get Normalized Mantissa from float32 xmm3/m32 and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

The conversion operation is:

$$GetMant(x) = \pm 2^{k}|x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then k = -1, otherwise K = 0. The encoded value of imm8[1:0] and sc are shown in Figure 5-23.

The converted DP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows the table below when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Table 5-17. 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)	If (SC[1]) then #IE

Below is a graphic view of imm8.

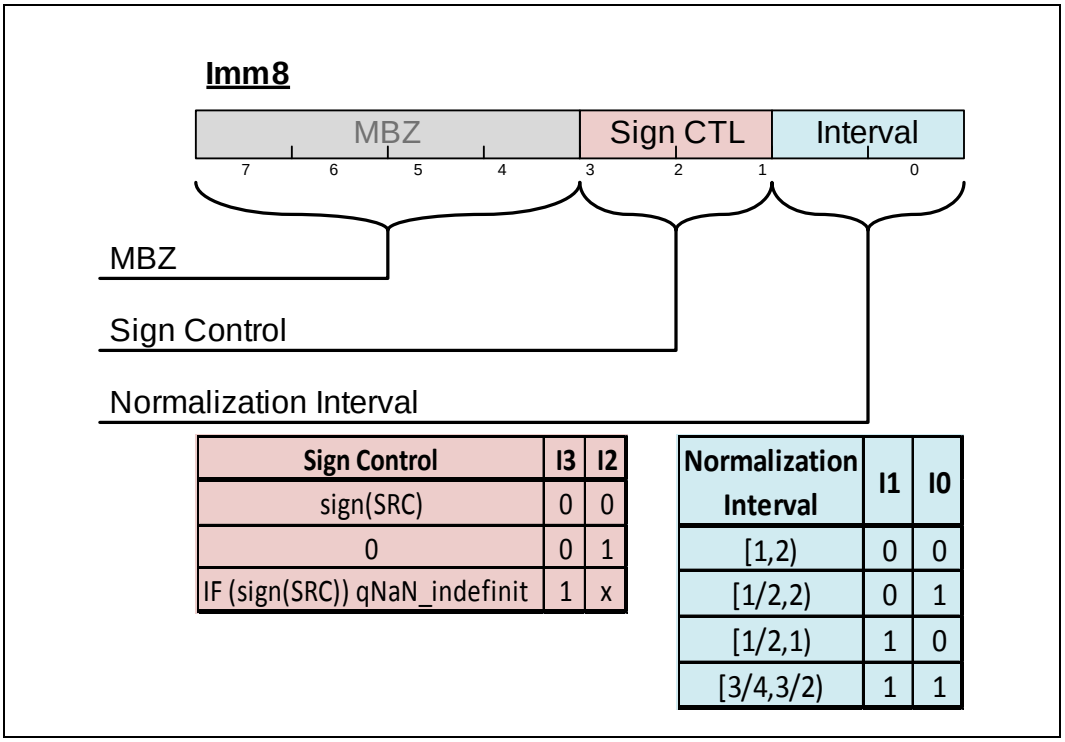


Figure 5-25. Graphic View of Imm8

Operation

```
GetNormalizeMantissa(SRC[31:0], SignCtrl[1:0], Interv[1:0])
{

    ; Extracting the SRC sign, exponent and mantissa fields
    Dst.sign ← SignCtrl[0] ? 0 : Src[31];          ; Get sign bit
Dst.exp ← SRC[30:23];                      ; Get original exponent value
```

```
Dst.fraction ← SRC[22:0];                ; Get original fraction value
ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
InfiniteOperand ← (Dst.exp = 0FFh) AND (Dst.fraction = 0);
NaNOperand ← (Dst.exp = 0FFh) AND (Dst.fraction != 0);

    ; Check for NAN operand
    IF (NaNOperand) THEN {
    IF (SRC = SNaN) THEN {Set #IE}
RETURN QNAN(SRC);
    }

    ; Check for Zero and Infinite operands
    IF ((ZeroOperand) OR (InfiniteOperand) THEN
{
    Dst.exp ← 07Fh;                      ; Override exponent with BIAS
RETURN ((Dst.sign<<31) | (Dst.exp<<23) | (Dst.fraction));
    }

    ; Check for negative operand (including -0.0)
    IF ((Src[31] = 1) AND SignCtrl[1]) THEN
{
Set #IE;
RETURN QNaN_Indefinite;
    }

; Checking for denormal operands
IF (DenormOperand) THEN {
    IF (MXCSR.DAZ=1) {
Dst.fraction ← 0;                        ; Zero out fraction
    }
    ELSE {
        ; Jbit is the hidden integral bit. Zero in case of denormal operand.
Src.Jbit ← 0;                            ; Zero Src Jbit
        Dst.exp ← 07Fh;                  ; Override exponent with BIAS
        WHILE (Src.Jbit = 0)             ; normalize mantissa
        {
            Src.Jbit ← Dst.fraction[22];         ; Get the fraction MSB
Dst.fraction ← (Dst.fraction << 1);          ; Start normalizing the mantissa
            Dst.exp-- ;              ; Adjust the exponent
    }
    SET #DE;             ; Set DE bit
}
}       ; At this point Dst.fraction is normalized.

; Checking for exponent response
Unbiased.exp ← Dst.exp – 07Fh;           ; subtract the bias from exponent
IsOddExp ← Unbiased.exp[0];              ; recognized unbiased ODD exponent
SignalingBit ← Dst.fraction[22];

CASE (interv[1:0]) of
    00: Dst.exp ← 07Fh;                  ; This is the bias
    01: Dst.exp ← (IsOddExp) ? 07Eh : 07Fh;          ; either bias-1, or bias
    10: Dst.exp ← 07Eh;                  ; bias-1
```

```
    11: Dst.exp ← (SignalingBit) ? 07Eh : 07Fh;                ; either bias-1, or bias
; At this point Dst.exp has the correct result.

; Form the final destination
DEST[31:0] ← (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);

RETURN (DEST);
}

SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];
```

VGETMANTSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ←
            GetNormalizedMantissa(SRC2[31:0], sc, interv)
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] ← 0
        FI
FI;
DEST[127:32] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTSS __m128 _mm_getmant_ss(__m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_mask_getmant_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_maskz_getmant_ss(__mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_getmant_round_ss(__m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_mask_getmant_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_maskz_getmant_round_ss(__mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Exceptions Type E3.

VINSERTF128/VINSERTF32x4/VINSERTF64x4—Insert Packed Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 18 /r ib VINSERTF128 ymm1, ymm2, xmm3/m128, imm8	RVMI	V/V	AVX	Insert 128-bits of packed floating-point values from xmm3/mem and the remaining values from ymm2 into ymm1.
EVEX.NDS.512.66.0F3A.W0 18 /r ib VINSERTF32x4 zmm1 {k1}{z}, zmm2, xmm3/mV, imm8	T4	V/V	AVX512F	Insert 128 bits of packed single-precision floating-point values from xmm3/mV and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W1 1A /r ib VINSERTF64x4 zmm1 {k1}{z}, zmm2, ymm3/mV, imm8	T4	V/V	AVX512F	Insert 256 bits of packed double-precision floating-point values from ymm3/mV and the remaining values from zmm2 into zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
T4	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

VINSERTF128/VINSERTF32x4 inserts128-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[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 high 6 bits of the immediate are ignored. The destination and first source operands are vector registers.

VINSERTF32x4: The destination operand is a ZMM register and updated at 32-bit granularity according to the writemask.

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 64-bit granularity according to the writemask.

Operation

VINSERTF32x4 (EVEX encoded versions)

```
(KL, VL) = (16, 512)
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
    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.
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
```

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
```

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);
VINSERTF64x4 __m512d _mm512_insertf64x4(__m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d _mm512_mask_insertf64x4(__m512d s, __mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d _mm512_maskz_insertf64x4(__mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF128 __m256 _mm256_insertf128_ps (__m256 a, __m128 b, int offset);
VINSERTF128 __m256d _mm256_insertf128_pd (__m256d a, __m128d b, int offset);
VINSERTF128 __m256i _mm256_insertf128_si256 (__m256i a, __m128i b, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instruction, see Exceptions Type 6; additionally

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E6NF.

VINSERTI128/VINSERTI32x4/VINSERTI64x4—Insert Packed Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 38 /r ib VINSERTI128 ymm1, ymm2, xmm3/m128, imm8	RVMI	V/V	AVX2	Insert 128-bits of integer data from xmm3/mem and the remaining values from ymm2 into ymm1.
EVEX.NDS.512.66.0F3A.W0 38 /r ib VINSERTI32x4 zmm1 {k1}{z}, zmm2, xmm3/mV, imm8	T4	V/V	AVX512F	Insert 128 bits of packed doubleword integer values from xmm3/mV and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W1 3A /r ib VINSERTI64x4 zmm1 {k1}{z}, zmm2, ymm3/mV, imm8	T4	V/V	AVX512F	Insert 256 bits of packed quadword integer values from ymm3/mV and the remaining values from zmm2 into zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
T4	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

VINSERTI32x4 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[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 bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32-bit granularity according to the writemask.

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 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) = (16, 512)
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
    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.
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
```

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
```

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 _mm512_inserti32x4(__m512i a, __m128i b, int imm);
VINSERTI32x4 _mm512_mask_inserti32x4(__m512i s, __mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 _mm512_maskz_inserti32x4(__mmask16 k, __m512i 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 Exceptions Type 6; additionally

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E6NF.

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	RMI	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.NDS.128.66.0F3A.WIG 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	RVMI	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.NDS.128.66.0F3A.W0 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	T1S	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

(register source form)

Select a single-precision floating-point element from second source as indicated by Count_S bits of the immediate operand 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.

(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 (255:128) of the corresponding ZMM 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 (MAX_VL-1:128) of the corresponding ZMM 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[MAX_VL-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[MAX_VL-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 Exceptions Type 5; additionally

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E9NF.

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	RM	V/V	SSE2	Return the maximum double-precision floating-point values between xmm1 and xmm2/mem.
VEX.NDS.128.66.0F.WIG 5F /r VMAXPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Return the maximum double-precision floating-point values between xmm2 and xmm3/mem.
VEX.NDS.256.66.0F.WIG 5F /r VMAXPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the maximum packed double-precision floating-point values between ymm2 and ymm3/mem.
EVEX.NDS.512.66.0F.W1 5F /r VMAXPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	FV	V/V	AVX512F	Return the maximum packed double-precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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, that 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 register. The second 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 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 (MAX_VL-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 (MAX_VL-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 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMAXPD (EVEX encoded versions)

```
(KL, VL) = (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
```

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])
```

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[MAX_VL-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[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

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_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 Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

MAXPS—Maximum of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 5F /r MAXPS xmm1, xmm2/m128	RM	V/V	SSE	Return the maximum single-precision floating-point values between xmm1 and xmm2/mem.
VEX.NDS.128.0F.WIG 5F /r VMAXPS xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Return the maximum single-precision floating-point values between xmm2 and xmm3/mem.
VEX.NDS.256.0F.WIG 5F /r VMAXPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the maximum single double-precision floating-point values between ymm2 and ymm3/mem.
EVEX.NDS.512.0F.W0 5F /r VMAXPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	FV	V/V	AVX512F	Return the maximum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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, that 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 register. 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 destination operand is a ZMM 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 (MAX_VL-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 (MAX_VL-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 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMAXPS (EVEX encoded versions)

```
(KL, VL) = (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
```

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])
```

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[MAX_VL-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[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

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_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 Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

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	RM	V/V	SSE2	Return the maximum scalar double-precision floating-point value between xmm2/m64 and xmm1.
VEX.NDS.128.F2.0F.WIG 5F /r VMAXSD xmm1, xmm2, xmm3/m64	RVM	V/V	AVX	Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.
EVEX.NDS.LIG.F2.0F.W1 5F /r VMAXSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	T1S	V/V	AVX512F	Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Compares the low double-precision floating-point values in the first source operand and second the 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 (MAX_VL-1:64) of the corresponding YMM 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 (MAX_VL-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 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 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) 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[MAX_VL-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[MAX_VL-1:128] ←0
```

MAXSD (128-bit Legacy SSE version)

```
DEST[63:0] ←MAX(DEST[63:0], SRC[63:0])
DEST[MAX_VL-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 Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.

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	RM	V/V	SSE	Return the maximum scalar single-precision floating-point value between xmm2/m32 and xmm1.
VEX.NDS.128.F3.0F.WIG 5F /r VMAXSS xmm1, xmm2, xmm3/m32	RVM	V/V	AVX	Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.
EVEX.NDS.LIG.F3.0F.W0 5F /r VMAXSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	T1S	V/V	AVX512F	Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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 (MAX_VL: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. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL: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 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) 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[MAX_VL-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[MAX_VL-1:128] ←0
```

MAXSS (128-bit Legacy SSE version)

```
DEST[31:0] ←MAX(DEST[31:0], SRC[31:0])
DEST[MAX_VL-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 Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.

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	RM	V/V	SSE2	Return the minimum double-precision floating-point values between xmm1 and xmm2/mem
VEX.NDS.128.66.0F.WIG 5D /r VMINPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Return the minimum double-precision floating-point values between xmm2 and xmm3/mem.
VEX.NDS.256.66.0F.WIG 5D /r VMINPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the minimum packed double-precision floating-point values between ymm2 and ymm3/mem.
EVEX.NDS.512.66.0F.W1 5D /r VMINPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	FV	V/V	AVX512F	Return the minimum packed double-precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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, that 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 register. The second 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 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 (MAX_VL-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 (MAX_VL-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 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
```

```
        ELSE IF (SRC1 < SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMINPD (EVEX encoded version)

```
(KL, VL) = (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
```

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[MAX_VL-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[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

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_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 Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

MINPS—Minimum of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 5D /r MINPS xmm1, xmm2/m128	RM	V/V	SSE	Return the minimum single-precision floating-point values between xmm1 and xmm2/mem.
VEX.NDS.128.0F.WIG 5D /r VMINPS xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Return the minimum single-precision floating-point values between xmm2 and xmm3/mem.
VEX.NDS.256.0F.WIG 5D /r VMINPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the minimum single double-precision floating-point values between ymm2 and ymm3/mem.
EVEX.NDS.512.0F.W0 5D /r VMINPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	FV	V/V	AVX512F	Return the minimum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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, that 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 register. 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 destination operand is a ZMM 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 (MAX_VL-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 (MAX_VL-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 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMINPS (EVEX encoded version)

```
(KL, VL) = (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
```

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[MAX_VL-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[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

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_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 Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

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	RM	V/V	SSE2	Return the minimum scalar double-precision floating-point value between xmm2/m64 and xmm1.
VEX.NDS.128.F2.0F.WIG 5D /r VMINSD xmm1, xmm2, xmm3/m64	RVM	V/V	AVX	Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.
EVEX.NDS.LIG.F2.0F.W1 5D /r VMINSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	T1S	V/V	AVX512F	Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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, 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 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 (MAX_VL-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 (MAX_VL-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 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) 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[MAX_VL-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[MAX_VL-1:128] ←0
```

MINSD (128-bit Legacy SSE version)

```
DEST[63:0] ←MIN(SRC1[63:0], SRC2[63:0])
DEST[MAX_VL-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 Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.

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	RM	V/V	SSE	Return the minimum scalar single-precision floating-point value between xmm2/m32 and xmm1.
VEX.NDS.128.F3.0F.WIG 5D /r VMINSS xmm1,xmm2, xmm3/m32	RVM	V/V	AVX	Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.
EVEX.NDS.LIG.F3.0F.W0 5D /r VMINSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	T1S	V/V	AVX512F	Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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 (MAX_VL: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 EVEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL: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 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) 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[MAX_VL-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[MAX_VL-1:128] ←0
```

MINSS (128-bit Legacy SSE version)

```
DEST[31:0] ←MIN(SRC1[31:0], SRC2[31:0])
DEST[MAX_VL-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 Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

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	RM	V/V	SSE2	Move aligned packed double-precision floating-point values from xmm2/mem to xmm1.
66 0F 29 /r MOVAPD xmm2/m128, xmm1	MR	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	RM	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	MR	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	RM	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	MR	V/V	AVX	Move aligned packed double-precision floating-point values from ymm1 to ymm2/mem.
EVEX.512.66.0F.W1 28 /r VMOVAPD zmm1 {k1}{z}, zmm2/mV	FVM-RM	V/V	AVX512F	Move aligned packed double-precision floating-point values from zmm2/mV to zmm1 using writemask k1.
EVEX.512.66.0F.W1 29 /r VMOVAPD zmm2/mV {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move aligned packed double-precision floating-point values from zmm1 to zmm2/mV using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

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 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 (MAX_VL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAX_VL-1:128) of the destination ZMM register destination are zeroed.

Operation

VMOVAPD (EVEX encoded versions, register-copy form)

```
(KL, VL) = (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
```

VMOVAPD (EVEX encoded versions, store-form)

```
(KL, VL) = (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) = (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
```

VMOVAPD (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAX_VL-1:256] ← 0
```

VMOVAPD (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVAPD (VEX.128 encoded version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAX_VL-1:128] ← 0
```

MOVAPD (128-bit load- and register-copy- form Legacy SSE version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAX_VL-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);
MOVAPD __m256d _mm256_load_pd (double * p);
MOVAPD _mm256_store_pd(double * p, __m256d a);
MOVAPD __m128d _mm_load_ps (double * p);
MOVAPD _mm_store_ps(double * p, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E1.

MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 28 /r MOVAPS xmm1, xmm2/m128	RM	V/V	SSE	Move aligned packed single-precision floating-point values from xmm2/mem to xmm1.
0F 29 /r MOVAPS xmm2/m128, xmm1	MR	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	RM	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	MR	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	RM	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	MR	V/V	AVX	Move aligned packed single-precision floating-point values from ymm1 to ymm2/mem.
EVEX.512.0F.W0 28 /r VMOVAPS zmm1 {k1}{z}, zmm2/mV	FVM-RM	V/V	AVX512F	Move aligned packed single-precision floating-point values from zmm2/mV to zmm1 using writemask k1.
EVEX.512.0F.W0 29 /r VMOVAPS zmm2/mV {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move aligned packed single-precision floating-point values from zmm1 to zmm2/mV using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

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 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 (MAX_VL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAX_VL-1:128) of the destination ZMM register are zeroed.

Operation

VMOVAPS (EVEX encoded versions, register-copy form)

```
(KL, VL) = (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
```

VMOVAPS (EVEX encoded versions, store-form)

```
(KL, VL) = (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) = (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
```

VMOVAPS (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAX_VL-1:256] ← 0
```

VMOVAPS (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVAPS (VEX.128 encoded version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAX_VL-1:128] ← 0
```

MOVAPS (128-bit load- and register-copy- form Legacy SSE version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAX_VL-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);
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; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E1.

MOVD/MOVQ—Move Doubleword and Quadword

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 6E /r MOVD xmm1, r32/m32	MR	V/V	SSE2	Move doubleword from r/m32 to xmm1.
66 REX.W 0F 6E /r MOVQ xmm1, r64/m64	MR	V/N.E.	SSE2	Move quadword from r/m64 to xmm1.
VEX.128.66.0F.W0 6E /r VMOVD xmm1, r32/m32	MR	V/V	AVX	Move doubleword from r/m32 to xmm1.
VEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64	MR	V/N.E.	AVX	Move quadword from r/m64 to xmm1.
EVEX.128.66.0F.W0 6E /r VMOVD xmm1, r32/m32	T1S-RM	V/V	AVX512F	Move doubleword from r/m32 to xmm1.
EVEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64	T1S-RM	V/N.E.[1]	AVX512F	Move quadword from r/m64 to xmm1.
66 0F 7E /r MOVD r32/m32, xmm1	MR	V/V	SSE2	Move doubleword from xmm1 register to r/m32.
66 REX.W 0F 7E /r MOVQ r64/m64, xmm1	MR	V/N.E.	SSE2	Move quadword from xmm1 register to r/m64.
VEX.128.66.0F.W0 7E /r VMOVD r32/m32, xmm1	MR	V/V	AVX	Move doubleword from xmm1 register to r/m32.
VEX.128.66.0F.W1 7E /r VMOVQ r64/m64, xmm1	MR	V/N.E.	AVX	Move quadword from xmm1 register to r/m64.
EVEX.128.66.0F.W0 7E /r VMOVD r32/m32, xmm1	T1S-MR	V/V	AVX512F	Move doubleword from xmm1 register to r/m32.
EVEX.128.66.0F.W1 7E /r VMOVQ r64/m64, xmm1	T1S-MR	V/N.E.[1]	AVX512F	Move quadword from xmm1 register to r/m64.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T1S-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

MOVD/Q with XMM destination:

Moves a dword integer from the source operand and stores it in the low 32-bits of the destination XMM register. The upper bits of the destination are zeroed. The source operand can be a 32-bit register or 32-bit memory location.

128-bit Legacy SSE version: Bits (MAX_VL-1:128) of the corresponding YMM destination register remain unchanged. A REX.W prefix promotes this to copy qword integers.

VEX.128 encoded version: Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: Bits (MAX_VL-1:128) of the destination register are zeroed.

MOVD/Q with r32/m32 or r64/m64 destination:

Stores 32 (64) bits from the low bits of the source XMM register.

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 (Legacy SSE version when destination is an XMM register)
DEST[31:0] ← SRC[31:0]
DEST[127:32] ← 0H
DEST[MAX_VL-1:128] (Unmodified)

VMOVD (VEX-encoded version when destination is an XMM register)
DEST[31:0] ← SRC[31:0]
DEST[MAX_VL-1:32] ← 0H

VMOVD (EVEX-encoded version when destination is an XMM register)
DEST[31:0] ← SRC[31:0]
DEST[511:32] ← 0H

MOVQ (Legacy SSE version when destination is an XMM register)
DEST[63:0] ← SRC[63:0]
DEST[127:64] ← 0H
DEST[MAX_VL-1:128] (Unmodified)

VMOVQ (VEX-encoded version when destination is an XMM register)
DEST[63:0] ← SRC[63:0]
DEST[MAX_VL-1:64] ← 0H

VMOVQ (EVEX-encoded version when destination is an XMM register)
DEST[63:0] ← SRC[63:0]
DEST[511:64] ← 0H

MOVD / VMOVD (when destination is not an XMM register)
DEST[31:0] ← SRC[31:0]

MOVQ / VMOVQ (when destination is not an XMM register)
DEST[63:0] ← SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

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);
MOVD __m128i _mm_cvtsi32_si128(int a)
MOVD int _mm_cvtsi128_si32(__m128i a)
MOVQ __m128i _mm_cvtsi64_si128(__int64 a)
MOVQ __int64 _mm_cvtsi128_si64(__m128i a)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E9NF.

MOVQ—Move Quadword

Opcode Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 7E /r MOVQ xmm1, xmm2/m64	RM	V/V	SSE2	Move quadword from xmm2/m64 to xmm1.
VEX.128.F3.0F.WIG 7E /r VMOVQ xmm1, xmm2	RM	V/V	AVX	Move quadword from xmm2 to xmm1.
VEX.128.F3.0F.WIG 7E /r VMOVQ xmm1, m64	RM	V/V	AVX	Load quadword from m64 to xmm1.
EVEX.128.F3.0F.W1 7E /r VMOVQ xmm1, xmm2	RM	V/V	AVX512F	Move quadword from xmm2 to xmm1.
EVEX.128.F3.0F.W1 7E /r VMOVQ xmm1, m64	T1S-RM	V/V	AVX512F	Load quadword from m64 to xmm1.
66 0F D6 /r MOVQ xmm1/m64, xmm2	MR	V/V	SSE2	Move quadword from xmm2 register to xmm1/m64.
VEX.128.66.0F D6.WIG /r VMOVQ xmm1/m64, xmm2	MR	V/V	AVX	Move quadword from xmm2 register to xmm1/m64.
EVEX.128.66.0F.W1 D6 /r VMOVQ xmm1/m64, xmm2	T1S-MR	V/V	AVX512F	Move quadword from xmm2 register to xmm1/m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T1S-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Copies a quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be an XMM register or a 64-bit memory locations. This instruction can be used 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.

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 (F3 0F 7E and 66 0F D6) with XMM register source and destination:
DEST[63:0] ← SRC[63:0]
DEST[127:64] ← 0
DEST[MAX_VL-1:128] (Unmodified)

VMOVQ (VEX.NDS.128.F3.0F 7E) with XMM register source and destination:
DEST[63:0] ← SRC[63:0]
DEST[MAX_VL-1:64] ← 0

VMOVQ (VEX.128.66.0F D6) with XMM register source and destination:
DEST[63:0] ← SRC[63:0]
DEST[MAX_VL-1:64] ← 0

VMOVQ (7E - EVEX encoded version) with XMM register source and destination:
DEST[63:0] ← SRC[63:0]
DEST[511:64] ← 0

VMOVQ (D6 - EVEX encoded version) with XMM register source and destination:
DEST[63:0] ← SRC[63:0]
DEST[511:64] ← 0

MOVQ (7E) with memory source:
DEST[63:0] ← SRC[63:0]
DEST[127:64] ← 0
DEST[MAX_VL-1:128] (Unmodified)

VMOVQ (7E - VEX.128 encoded version) with memory source:
DEST[63:0] ← SRC[63:0]
DEST[MAX_VL-1:64] ← 0

VMOVQ (7E - EVEX encoded version) with memory source:
DEST[63:0] ← SRC[63:0]
DEST[511:64] ← 0

MOVQ (D6) with memory dest:
DEST[63:0] ← SRC[63:0]

VMOVQ (D6) with memory dest:
DEST[63:0] ← SRC2[63:0]

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

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E9NF.

MOVDDUP—Replicate Double FP Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 12 /r MOVDDUP xmm1, xmm2/m64	RM	V/V	SSE3	Move double-precision floating-point values from xmm2/mem and duplicate into xmm1.
VEX.128.F2.0F.WIG 12 /r VMOVDDUP xmm1, xmm2/m64	RM	V/V	AVX	Move double-precision floating-point values from xmm2/mem and duplicate into xmm1.
VEX.256.F2.0F.WIG 12 /r VMOVDDUP ymm1, ymm2/m256	RM	V/V	AVX	Move even index double-precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.512.F2.0F.W1 12 /r VMOVDDUP zmm1 {k1}{z}, zmm2/m512	DUP-RM	V/V	AVX512F	Move even index double-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
DUP-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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).

128-bit Legacy SSE version: Bits (MAX_VL-1:128) of the corresponding destination register are unchanged. The source operand is XMM register or a 64-bit memory location.

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

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

EVEX 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.

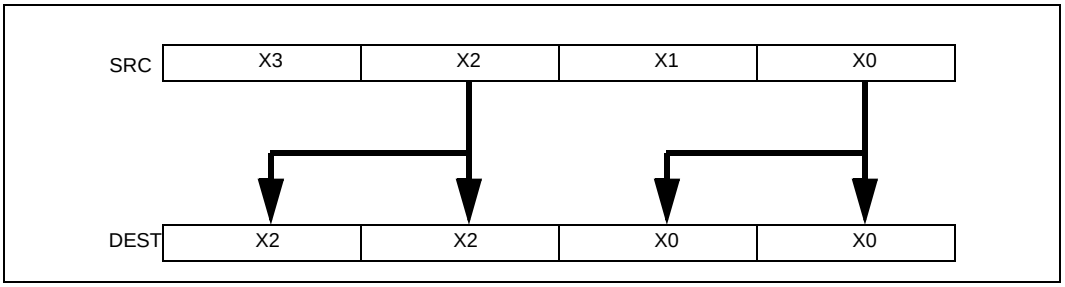


Figure 5-26. VMOVDDUP Operation

Operation

VMOVDDUP (EVEX encoded versions)

```
(KL, VL) = (8, 512)
TMP_SRC[63:0] ← SRC[63:0]
TMP_SRC[127:64] ← SRC[63:0]
    TMP_SRC[191:128] ← SRC[191:128]
    TMP_SRC[255:192] ← SRC[191:128]
    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]
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
```

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]
```

VMOVDDUP (VEX.128 encoded version)

```
DEST[63:0] ←SRC[63:0]
DEST[127:64] ←SRC[63:0]
DEST[MAX_VL-1:128] ←0
```

MOVDDUP (128-bit Legacy SSE version)

```
DEST[63:0] ←SRC[63:0]
DEST[127:64] ←SRC[63:0]
DEST[MAX_VL-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);
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 Exceptions Type 5; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E5NF.

MOVDQA—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	RM	V/V	SSE2	Move aligned packed integer values from xmm2/mem to xmm1.
66 0F 7F /r MOVDQA xmm2/m128, xmm1	MR	V/V	SSE2	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 6F /r VMOVDQA xmm1, xmm2/m128	RM	V/V	AVX	Move aligned packed integer values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 7F /r VMOVDQA xmm2/m128, xmm1	MR	V/V	AVX	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 6F /r VMOVDQA ymm1, ymm2/m256	RM	V/V	AVX	Move aligned packed integer values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 7F /r VMOVDQA ymm2/m256, ymm1	MR	V/V	AVX	Move aligned packed integer values from ymm1 to ymm2/mem.
EVEX.512.66.0F.W0 6F /r VMOVDQA32 zmm1 {k1}{z}, zmm2/mV	FVM-RM	V/V	AVX512F	Move aligned packed doubleword integer values from zmm2/mV to zmm1 using writemask k1.
EVEX.512.66.0F.W0 7F /r VMOVDQA32 zmm2/mV {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move aligned packed doubleword integer values from zmm1 to zmm2/mV using writemask k1.
EVEX.512.66.0F.W1 6F /r VMOVDQA64 zmm1 {k1}{z}, zmm2/mV	FVM-RM	V/V	AVX512F	Move aligned packed quadword integer values from zmm2/mV to zmm1 using writemask k1.
EVEX.512.66.0F.W1 7F /r VMOVDQA64 zmm2/mV {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move aligned packed quadword integer values from zmm1 to zmm2/mV using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

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

EVEX encoded versions:

Moves 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 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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAX_VL-1:128) of the destination register are zeroed.

Operation

VMOVDQA32 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (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
```

VMOVDQA32 (EVEX encoded versions, store-form)

```
(KL, VL) = (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) = (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
```

VMOVDQA64 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (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
```

VMOVDQA64 (EVEX encoded versions, store-form)

```
(KL, VL) = (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) = (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
```

VMOVDQA (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAX_VL-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[MAX_VL-1:128] ← 0

VMOVDQA (128-bit load- and register-copy- form Legacy SSE version)
DEST[127:0] ← SRC[127:0]
DEST[MAX_VL-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);
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);
MOVDQA __m256i _mm256_load_si256 (__m256i * p);
MOVDQA _mm256_store_si256(_m256i *p, __m256i a);
MOVDQA __m128i _mm_load_si128 (__m128i * p);
MOVDQA _mm_store_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E1.

MOVDQU/VMOVDQU32/VMOVDQU64—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	RM	V/V	SSE2	Move unaligned packed integer values from xmm2/mem to xmm1.
F3 0F 7F /r MOVDQU xmm2/m128, xmm1	MR	V/V	SSE2	Move unaligned packed integer values from xmm1 to xmm2/mem.
VEX.128.F3.0F.WIG 6F /r VMOVDQU xmm1, xmm2/m128	RM	V/V	AVX	Move unaligned packed integer values from xmm2/mem to xmm1.
VEX.128.F3.0F.WIG 7F /r VMOVDQU xmm2/m128, xmm1	MR	V/V	AVX	Move unaligned packed integer values from xmm1 to xmm2/mem.
VEX.256.F3.0F.WIG 6F /r VMOVDQU ymm1, ymm2/m256	RM	V/V	AVX	Move unaligned packed integer values from ymm2/mem to ymm1.
VEX.256.F3.0F.WIG 7F /r VMOVDQU ymm2/m256, ymm1	MR	V/V	AVX	Move unaligned packed integer values from ymm1 to ymm2/mem.
EVEX.512.F3.0F.W0 6F /r VMOVDQU32 zmm1 {k1}{z}, zmm2/mV	FVM-RM	V/V	AVX512F	Move unaligned packed doubleword integer values from zmm2/mV to zmm1 using writemask k1.
EVEX.512.F3.0F.W0 7F /r VMOVDQU32 zmm2/mV {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move unaligned packed doubleword integer values from zmm1 to zmm2/mV using writemask k1.
EVEX.512.F3.0F.W1 6F /r VMOVDQU64 zmm1 {k1}{z}, zmm2/mV	FVM-RM	V/V	AVX512F	Move unaligned packed quadword integer values from zmm2/mV to zmm1 using writemask k1.
EVEX.512.F3.0F.W1 7F /r VMOVDQU64 zmm2/mV {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move unaligned packed quadword integer values from zmm1 to zmm2/mV using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

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

EVEX encoded versions:

Moves 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 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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-1:128) of the destination register are zeroed.

Operation

VMOVDQU32 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (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
```

VMOVDQU32 (EVEX encoded versions, store-form)

```
(KL, VL) = (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) = (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
```

VMOVDQU64 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (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
```

VMOVDQU64 (EVEX encoded versions, store-form)

```
(KL, VL) = (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;
```

VMOVDQU64 (EVEX encoded versions, load-form)

```
(KL, VL) = (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
```

VMOVDQU (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAX_VL-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[MAX_VL-1:128] ← 0

VMOVDQU (128-bit load- and register-copy- form Legacy SSE version)
DEST[127:0] ← SRC[127:0]
DEST[MAX_VL-1:128] (Unmodified)

(V)MOVDQU (128-bit store-form version)
DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

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);
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);
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 Exceptions Type 4; additionally

#UD	If VEX.vvvv != 1111B.

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

MOVHLPS—Move Packed Single-Precision Floating-Point Values High to Low

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
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.NDS.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.NDS.128.0F.W0 12 /r VMOVHLPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F	Merge two packed single-precision floating-point values from high quadword of xmm3 and low quadword of xmm2.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	vvvv (r)	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-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[MAX_VL-1:64] (Unmodified)
```

VMOVHLPS (128-bit three-argument form - VEX & EVEX)

```
DEST[63:0] ← SRC2[127:64]
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVHLPS __m128 _mm_movehl_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None

1. ModRM.MOD = 011B required

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 7; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128.

MOVHPD—Move High Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 16 /r MOVHPD xmm1, m64	RM	V/V	SSE2	Move double-precision floating-point values from m64 to high quadword of xmm1.
VEX.NDS.128.66.0F.WIG 16 /r VMOVHPD xmm2, xmm1, m64	RVM	V/V	AVX	Merge double-precision floating-point value from m64 and the low quadword of xmm1.
EVEX.NDS.128.66.0F.W1 16 /r VMOVHPD xmm2, xmm1, m64	T1S	V/V	AVX512F	Merge double-precision floating-point value from m64 and the low quadword of xmm1.
66 0F 17/r MOVHPD m64, xmm1	MR	V/V	SSE2	Move double-precision floating-point values from high quadword of xmm1 to m64.
VEX128.66.0F.WIG 17/r VMOVHPD m64, xmm1	MR	V/V	AVX	Move double-precision floating-point values from high quadword of xmm1 to m64.
EVEX.NDS.128.66.0F.W1 17/r VMOVHPD m64, xmm1	T1S-MR	V/V	AVX512F	Move double-precision floating-point values from high quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

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 (MAX_VL-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 (MAX_VL-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[MAX_VL-1:128] (Unmodified)

VMOVHPD (VEX.128 & EVEX encoded load)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAX_VL-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 Exceptions Type 5; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

MOVHPS—Move High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 16 /r MOVHPS xmm1, m64	RM	V/V	SSE	Move two packed single-precision floating-point values from m64 to high quadword of xmm1.
VEX.NDS.128.0F.WIG 16 /r VMOVHPS xmm2, xmm1, m64	RVM	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.
EVEX.NDS.128.0F.W0 16 /r VMOVHPS xmm2, xmm1, m64	T1S	V/V	AVX512F	Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.
0F 17/r MOVHPS m64, xmm1	MR	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	MR	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	T1S-MR	V/V	AVX512F	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

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 (MAX_VL-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 (MAX_VL-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.NDS.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[MAX_VL-1:128] (Unmodified)

VMOVHPS (VEX.128 and EVEX encoded load)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAX_VL-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 Exceptions Type 5; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
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.NDS.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.NDS.128.0F.W0 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F	Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	vvvv (r)	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding destination register are zeroed.

If VMOVLHPS is encoded with VEX.Lor 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[MAX_VL-1:128] (Unmodified)
```

VMOVLHPS (128-bit three-argument form - VEX & EVEX)

```
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAX_VL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVLHPS __m128 _mm_movelh_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None

1. ModRM.MOD = 011B required

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 7; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128.

MOVLPD—Move Low Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 12 /r MOVLPD xmm1, m64	RM	V/V	SSE2	Move double-precision floating-point values from m64 to low quadword of xmm1.
VEX.NDS.128.66.0F.WIG 12 /r VMOVLPD xmm2, xmm1, m64	RVM	V/V	AVX	Merge double-precision floating-point value from m64 and the high quadword of xmm1.
EVEX.NDS.128.66.0F.W1 12 /r VMOVLPD xmm2, xmm1, m64	T1S	V/V	AVX512F	Merge double-precision floating-point value from m64 and the high quadword of xmm1.
66 0F 13/r MOVLPD m64, xmm1	MR	V/V	SSE2	Move double-precision floating-point values from low quadword of xmm1 to m64.
VEX.128.66.0F.WIG 13/r VMOVLPD m64, xmm1	MR	V/V	AVX	Move double-precision floating-point values from low quadword of xmm1 to m64.
EVEX.128.66.0F.W1 13/r VMOVLPD m64, xmm1	T1S-MR	V/V	AVX512F	Move double-precision floating-point values from low quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:r/m (r)	VEX.vvvv	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

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 (MAX_VL-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 (MAX_VL-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[MAX_VL-1:64] (Unmodified)

VMOVLPD (VEX.128 & EVEX encoded load)
DEST[63:0] ← SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-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 Exceptions Type 5; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

MOVLPS—Move Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 12 /r MOVLPS xmm1, m64	RM	V/V	SSE	Move two packed single-precision floating-point values from m64 to low quadword of xmm1.
VEX.NDS.128.0F.WIG 12 /r VMOVLPS xmm2, xmm1, m64	RVM	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.
EVEX.NDS.128.0F.W0 12 /r VMOVLPS xmm2, xmm1, m64	T1S	V/V	AVX512F	Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.
0F 13/r MOVLPS m64, xmm1	MR	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	MR	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	T1S-MR	V/V	AVX512F	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

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 (MAX_VL-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 (MAX_VL-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[MAX_VL-1:64] (Unmodified)

VMOVLPS (VEX.128 & EVEX encoded load)
DEST[63:0] ← SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-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 Exceptions Type 5; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

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	RM	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	RM	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	RM	V/V	AVX2	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	FVM	V/V	AVX512F	Move 512-bit data from m512 to zmm using non-temporal hint if WC memory type.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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 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 implemen-

1. ModRM.MOD = 011B required

tation 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[MAX_VL-1:128] (Unmodified)

VMOVNTDQA (VEX.128 encoded form)
DEST ← SRC
DEST[MAX_VL-1:128] ← 0

VMOVNTDQA (VEX.256 encoded forms)
DEST[255:0] ← SRC[255:0]
DEST[MAX_VL-1:256] ← 0

VMOVNTDQA (EVEX.512 encoded form)
DEST[511:0] ← SRC[511:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTDQA __m512i _mm512_stream_load_si512(void * p);
MOVNTDQA __m128i _mm_stream_load_si128 (__m128i *p);
VMOVNTDQA __m256i _mm_stream_load_si256 (__m256i *p);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E1NF.

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	MR	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	MR	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	MR	V/V	AVX	Move packed integer values in ymm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W0 E7 /r VMOVNTDQ m512, zmm1	FVM	V/V	AVX512F	Move packed integer values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

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, double-words, 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.

Operation

VMOVNTDQ(EVEX encoded versions)

```
VL = 512
DEST[VL-1:0] ← SRC[VL-1:0]
```

MOVNTDQ (Legacy and VEX versions)

```
DEST ← SRC
```

1. ModRM.MOD = 011B required

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; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E1NF.

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	MR	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	MR	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	MR	V/V	AVX	Move packed double-precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W1 2B /r VMOVNTPD m512, zmm1	FVM	V/V	AVX512F	Move packed double-precision values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

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.

Operation

VMOVNTPD (EVEX encoded versions)

```
VL = 512
DEST[VL-1:0] ← SRC[VL-1:0]
```

MOVNTPD (Legacy and VEX versions)

```
DEST ← SRC
```

1. ModRM.MOD = 011B required

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; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E1NF.

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
0F 2B /r MOVNTPS m128, xmm1	MR	V/V	SSE	Move packed single-precision values xmm1 to mem using non-temporal hint.
VEX.128.0F.WIG 2B /r VMOVNTPS m128, xmm1	MR	V/V	AVX	Move packed single-precision values xmm1 to mem using non-temporal hint.
VEX.256.0F.WIG 2B /r VMOVNTPS m256, ymm1	MR	V/V	AVX	Move packed single-precision values ymm1 to mem using non-temporal hint.
EVEX.512.66.0F.W0 2B /r VMOVNTPS m512, zmm1	FVM	V/V	AVX512F	Move packed single-precision values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

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.

Operation

VMOVNTPS (EVEX encoded versions)

```
VL = 512
DEST[VL-1:0] ← SRC[VL-1:0]
```

MOVNTPS

```
DEST ← SRC
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTPS void _mm512_stream_ps(float * p, __m512d a);

1. ModRM.MOD = 011B required

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; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E1NF.

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	RM	V/V	SSE2	Merge or Move scalar double-precision floating-point value from xmm2 to xmm1 register.
F2 0F 10 /r MOVSD xmm1, m64	RM	V/V	SSE2	Merge or Move scalar double-precision floating-point value from m64 to xmm1 register.
F2 0F 11 /r MOVSD xmm1/m64, xmm2	MR	V/V	SSE2	Move scalar double-precision floating-point value from xmm2 register to xmm1/m64.
VEX.NDS.LIG.F2.0F.WIG 10 /r VMOVSD xmm1, xmm2, xmm3	RVM	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	XM	V/V	AVX	Load scalar double-precision floating-point value from m64 to xmm1 register.
VEX.NDS.LIG.F2.0F.WIG 11 /r VMOVSD xmm1, xmm2, xmm3	MVR	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	MR	V/V	AVX	Move scalar double-precision floating-point value from xmm1 register to m64.
EVEX.NDS.LIG.F2.0F.W1 10 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	RVM	V/V	AVX512F	Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LIG.F2.0F.W1 10 /r VMOVSD xmm1 {k1}{z}, m64	T1S-RM	V/V	AVX512F	Load scalar double-precision floating-point value from mV to xmm1 register under writemask k1.
EVEX.NDS.LIG.F2.0F.W1 11 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	MVR	V/V	AVX512F	Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LIG.F2.0F.W1 11 /r VMOVSD m64 {k1}, xmm1	T1S-MR	V/V	AVX512F	Move scalar double-precision floating-point value from xmm1 register to mV under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
XM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MVR	ModRM:r/m (w)	vvvv (r)	ModRM:reg (r)	NA
T1S-RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

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 MAX_VL: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 MAX_VL: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 (MAX_VL-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 MAX_VL: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.NDS.LIG.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[511:64] ← 0
```

VMOVSD (EVEX.NDS.LIG.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.NDS.LIG.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[MAX_VL-1:128] ← 0
```

MOVSD (128-bit Legacy SSE version: MOVSD XMM1, XMM2)
DEST[63:0] ←SRC[63:0]
DEST[MAX_VL-1:64] (Unmodified)

VMOVSD (VEX.NDS.128.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)
DEST[63:0] ←SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAX_VL-1:128] ←0

VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)
DEST[63:0] ←SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAX_VL-1:128] ←0

VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, m64)
DEST[63:0] ←SRC[63:0]
DEST[MAX_VL-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[MAX_VL-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 Exceptions Type 5; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E10.

MOVSHDUP—Replicate Single FP Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 16 /r MOVSHDUP xmm1, xmm2/m128	RM	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	RM	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	RM	V/V	AVX	Move odd index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.512.F3.0F.W0 16 /r VMOVSHDUP zmm1 {k1}{z}, zmm2/m512	FVM	V/V	AVX512F	Move odd index single-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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 5-27. 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, YMMM or ZMM register.

128-bit Legacy SSE version: Bits (MAX_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAX_VL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAX_VL-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.

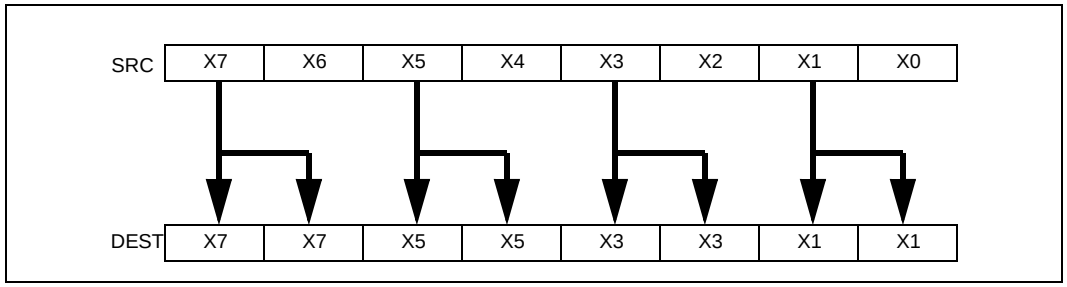


Figure 5-27. MOVSHDUP Operation

Operation

VMOVSHDUP (EVEX encoded versions)

(KL, VL) = (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]
    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]
    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]
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
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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_movehdup_ps (__m256 a);
VMOVSHDUP __m128 _mm_movehdup_ps (__m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

MOVSLDUP—Replicate Single FP 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	RM	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	RM	V/V	AVX	Move even index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.512.F3.0F.W0 12 /r VMOVSLDUP zmm1 {k1}{z}, zmm2/mV	FVM	V/V	AVX512F	Move even index single-precision floating-point values from zmm2/mV and duplicate each element into zmm1 under writemask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Duplicates even-indexed single-precision floating-point values from the source operand (the second operand). See Figure 5-28. 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 (MAX_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAX_VL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAX_VL-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.

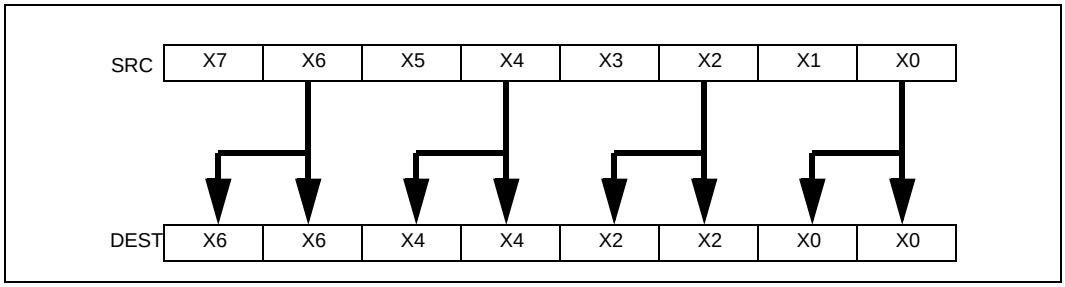


Figure 5-28. MOVSLDUP Operation

Operation

VMOVSLDUP (EVEX encoded versions)

(KL, VL) = (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]
    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]
    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]
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
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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_moveldup_ps (__m256 a);
VMOVSLDUP __m128 _mm_moveldup_ps (__m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

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	RM	V/V	SSE	Merge scalar single-precision floating-point value from xmm2 to xmm1 register.
F3 0F 10 /r MOVSS xmm1, m32	RM	V/V	SSE	Load scalar single-precision floating-point value from m32 to xmm1 register.
VEX.NDS.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, xmm2, xmm3	RVM	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	XM	V/V	AVX	Load scalar single-precision floating-point value from m32 to xmm1 register.
F3 0F 11 /r MOVSS xmm2/m32, xmm1	MR	V/V	SSE	Move scalar single-precision floating-point value from xmm1 register to xmm2/m32.
VEX.NDS.LIG.F3.0F.WIG 11 /r VMOVSS xmm1, xmm2, xmm3	MVR	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	MR	V/V	AVX	Move scalar single-precision floating-point value from xmm1 register to m32.
EVEX.NDS.LIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	RVM	V/V	AVX512F	Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, m32	T1S-RM	V/V	AVX512F	Move scalar single-precision floating-point values from m32 to xmm1 under writemask k1.
EVEX.NDS.LIG.F3.0F.W0 11 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	MVR	V/V	AVX512F	Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LIG.F3.0F.W0 11 /r VMOVSS m32 {k1}, xmm1	T1S-MR	V/V	AVX512F	Move scalar single-precision floating-point values from xmm1 to m32 under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
XM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MVR	ModRM:r/m (w)	vvvv (r)	ModRM:reg (r)	NA
T1S-RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

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 (MAX_VL-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 MAX_VL: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 (MAX_VL-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 MAX_VL: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.NDS.LIG.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[511:32] ← 0
```

VMOVSS (EVEX.NDS.LIG.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.NDS.LIG.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[MAX_VL-1:128] ← 0

MOVSS (Legacy SSE version when the source and destination operands are both XMM registers)
DEST[31:0] ←SRC[31:0]
DEST[MAX_VL-1:32] (Unmodified)

VMOVSS (VEX.NDS.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[MAX_VL-1:128] ←0

VMOVSS (VEX.NDS.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[MAX_VL-1:128] ←0

VMOVSS (VEX.NDS.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[MAX_VL-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[MAX_VL-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 Exceptions Type 5; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E10.

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	RM	V/V	SSE2	Move unaligned packed double-precision floating-point from xmm2/mem to xmm1.
66 0F 11 /r MOVUPD xmm2/m128, xmm1	MR	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	RM	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	MR	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	RM	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	MR	V/V	AVX	Move unaligned packed double-precision floating-point from ymm1 to ymm2/mem.
EVEX.512.66.0F.W1 10 /r VMOVUPD zmm1 {k1}{z}, zmm2/mV	FVM-RM	V/V	AVX512F	Move unaligned packed double-precision floating-point values from zmm2/mV to zmm1 using writemask k1.
EVEX.512.66.0F.W1 11 /r VMOVUPD zmm2/mV {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move unaligned packed double-precision floating-point values from zmm1 to zmm2/mV using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

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 (MAX_VL-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 (MAX_VL-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 encoded versions: Bits (MAX_VL-1:128) of the destination register are zeroed.

Operation

VMOVUPD (EVEX encoded versions, register-copy form)

```
(KL, VL) = (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
```

VMOVUPD (EVEX encoded versions, store-form)

```
(KL, VL) = (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;
```

VMOVUPD (EVEX encoded versions, load-form)

```
(KL, VL) = (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
```

VMOVUPD (VEX.256 encoded version, load - and register copy)
DEST[255:0] ← SRC[255:0]
DEST[MAX_VL-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[MAX_VL-1:128] ← 0

MOVUPD (128-bit load- and register-copy- form Legacy SSE version)
DEST[127:0] ← SRC[127:0]
DEST[MAX_VL-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);
MOVUPD __m256d _mm256_loadu_pd (__m256d * p);
MOVUPD _mm256_storeu_pd(_m256d *p, __m256d a);
MOVUPD __m128d _mm_loadu_pd (__m128d * p);
MOVUPD _mm_storeu_pd(__m128d *p, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
Note treatment of #AC varies; additionally
#UD If VEX.vvvv != 1111B.
EVEX-encoded instruction, see Exceptions Type E4.nb.

MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 10 /r MOVUPS xmm1, xmm2/m128	RM	V/V	SSE	Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.
0F 11 /r MOVUPS xmm2/m128, xmm1	MR	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	RM	V/V	AVX	Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.
VEX.128.0F 11.WIG /r VMOVUPS xmm2/m128, xmm1	MR	V/V	AVX	Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem.
VEX.256.0F 10.WIG /r VMOVUPS ymm1, ymm2/m256	RM	V/V	AVX	Move unaligned packed single-precision floating-point from ymm2/mem to ymm1.
VEX.256.0F 11.WIG /r VMOVUPS ymm2/m256, ymm1	MR	V/V	AVX	Move unaligned packed single-precision floating-point from ymm1 to ymm2/mem.
EVEX.512.0F.W0 10 /r VMOVUPS zmm1 {k1}{z}, zmm2/mV	FVM-RM	V/V	AVX512F	Move unaligned packed single-precision floating-point values from zmm2/mV to zmm1 using writemask k1.
EVEX.512.0F.W0 11 /r VMOVUPS zmm2/mV {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move unaligned packed single-precision floating-point values from zmm1 to zmm2/mV using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

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 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 (MAX_VL-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 (MAX_VL-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 encoded versions: Bits (MAX_VL-1:128) of the destination register are zeroed.

Operation

VMOVUPS (EVEX encoded versions, register-copy form)

```
(KL, VL) = (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
```

VMOVUPS (EVEX encoded versions, store-form)

```
(KL, VL) = (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) = (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
```

VMOVUPS (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAX_VL-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[MAX_VL-1:128] ← 0

MOVUPS (128-bit load- and register-copy- form Legacy SSE version)
DEST[127:0] ← SRC[127:0]
DEST[MAX_VL-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);
MOVUPS __m256 _mm256_loadu_ps (__m256 * p);
MOVUPS _mm256_storeu_ps(_m256 *p, __m256 a);
MOVUPS __m128 _mm_loadu_ps (__m128 * p);
MOVUPS _mm_storeu_ps(__m128 *p, __m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
Note treatment of #AC varies; additionally

#UD	If VEX.vvvv != 1111B.

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

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	RM	V/V	SSE2	Multiply packed double-precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.NDS.128.66.0F.WIG 59 /r VMULPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Multiply packed double-precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.NDS.256.66.0F.WIG 59 /r VMULPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Multiply packed double-precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.NDS.512.66.0F.W1 59 /r VMULPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values in zmm3/m512/m64bcst with zmm2 and store result in zmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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 register. The second 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 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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VMULPD (EVEX encoded versions)

```
(KL, VL) = (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
```

```
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← 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
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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 Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

MULPS—Multiply Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 59 /r MULPS xmm1, xmm2/m128	RM	V/V	SSE	Multiply packed single-precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.NDS.128.0F.WIG 59 /r VMULPS xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Multiply packed single-precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.NDS.256.0F.WIG 59 /r VMULPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Multiply packed single-precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.NDS.512.0F.W0 59 /r VMULPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values in zmm3/m512/m32bcst with zmm2 and store result in zmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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 register. 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 destination operand is a ZMM 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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VMULPS (EVEX encoded version)

```
(KL, VL) = (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
```

```
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← 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
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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_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 Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

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	RM	V/V	SSE2	Multiply the low double-precision floating-point value in xmm2/m64 by low double-precision floating-point value in xmm1.
VEX.NDS.128.F2.0F.WIG 59/r VMULSD xmm1,xmm2, xmm3/m64	RVM	V/V	AVX	Multiply the low double-precision floating-point value in xmm3/m64 by low double-precision floating-point value in xmm2.
EVEX.NDS.LIG.F2.0F.W1 59/r VMULSD xmm1 {k1}{z}, xmm2, xmm3/m64 {er}	T1S	V/V	AVX512F	Multiply the low double-precision floating-point value in xmm3/m64 by low double-precision floating-point value in xmm2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-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 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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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[MAX_VL-1:128] ← 0
```

VMULSD (VEX.128 encoded version)

```
DEST[63:0] ←SRC1[63:0] * SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAX_VL-1:128] ←0
```

MULSD (128-bit Legacy SSE version)

```
DEST[63:0] ←DEST[63:0] * SRC[63:0]
DEST[MAX_VL-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 Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.

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	RM	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.NDS.128.F3.0F.WIG 59 /r VMULSS xmm1,xmm2, xmm3/m32	RVM	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.NDS.LIG.F3.0F.W0 59 /r VMULSS xmm1 {k1}{z}, xmm2, xmm3/m32 {er}	T1S	V/V	AVX512F	Multiply the low single-precision floating-point value in xmm3/m32 by the low single-precision floating-point value in xmm2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-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 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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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[MAX_VL-1:128] ← 0
```

VMULSS (VEX.128 encoded version)

```
DEST[31:0] ←SRC1[31:0] * SRC2[31:0]
DEST[127:32] ←SRC1[127:32]
DEST[MAX_VL-1:128] ←0
```

MULSS (128-bit Legacy SSE version)

```
DEST[31:0] ←DEST[31:0] * SRC[31:0]
DEST[MAX_VL-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 Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.

PABSB/PABSW/PABSD/PABSQ—Packed Absolute Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 1C /r PABSB xmm1, xmm2/m128	RM	V/V	SSSE3	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
66 0F 38 1D /r PABSW xmm1, xmm2/m128	RM	V/V	SSSE3	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
66 0F 38 1E /r PABSD xmm1, xmm2/m128	RM	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	RM	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	RM	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	RM	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	RM	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	RM	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	RM	V/V	AVX2	Compute the absolute value of 32-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.
EVEX.512.66.0F38.W0 1E /r VPABSD zmm1 {k1}{z}, zmm2/m512/m32bcst	FV	V/V	AVX512F	Compute the absolute value of 32-bit integers in zmm2/m512/m32bcst and store UNSIGNED result in zmm1 using writemask k1.
EVEX.512.66.0F38.W1 1F /r VPABSQ zmm1 {k1}{z}, zmm2/m512/m64bcst	FV	V/V	AVX512F	Compute the absolute value of 64-bit integers in zmm2/m512/m64bcst and store UNSIGNED result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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 16-bit words, and PABSD operates on signed 32-bit integers.

EVEX.512: 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 updated according to the writemask.

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

VEX.128 versions: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-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])
```

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])
```

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) = (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;
```

VPABSQ (EVEX encoded versions)

```
(KL, VL) = (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;
```

Intel C/C++ Compiler Intrinsic Equivalents

```
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);
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 Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

PADDB/PADDW/PADDD/PADDQ—Add Packed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F FC /r PADDB *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed byte integers from *xmm2/m128* and *xmm1*.
66 0F FD /r PADDW *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed word integers from *xmm2/m128* and *xmm1*.
66 0F FE /r PADDD *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed doubleword integers from *xmm2/m128* and *xmm1*.
66 0F D4 /r PADDQ *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed quadword integers from *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG FC /r VPADDB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed byte integers from *xmm2, and xmm3/m128* and store in *xmm1*.
VEX.NDS.128.66.0F.WIG FD /r VPADDW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed word integers from *xmm2, xmm3/m128* and *store in xmm1*.
VEX.NDS.128.66.0F.WIG FE /r VPADDD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed doubleword integers from *xmm2, xmm3/m128* and store in *xmm1*.
VEX.NDS.128.66.0F.WIG D4 /r VPADDQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed quadword integers from *xmm2, xmm3/m128* and store in *xmm1*.
VEX.NDS.256.66.0F.WIG FC /r VPADDB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed byte integers from *ymm2, and ymm3/m256* and store in *ymm1*.
VEX.NDS.256.66.0F.WIG FD /r VPADDW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed word integers from *ymm2, ymm3/m256* and *store in ymm1*.
VEX.NDS.256.66.0F.WIG FE /r VPADDD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed doubleword integers from *ymm2, ymm3/m256* and *store in ymm1*.
VEX.NDS.256.66.0F.WIG D4 /r VPADDQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed quadword integers from *ymm2, ymm3/m256* and store in *ymm1*.
EVEX.NDS.512.66.0F.W0 FE /r VPADDD *zmm1 {k1}{z}, zmm2,* zmm3/m512/m32bcst	FV	V/V	AVX512F	Add packed doubleword integers from *zmm2, zmm3/m512/m32bcst* and store in *zmm1* using writemask k1.
EVEX.NDS.512.66.0F.W1 D4 /r VPADDQ *zmm1 {k1}{z}, zmm2,* zmm3/m512/m64bcst	FV	V/V	AVX512F	Add packed quadword integers from *zmm2, zmm3/m512/m64bcst* and store in *zmm1* using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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 versions: 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 32/64-bit memory location. The destination operand is a ZMM 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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

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];
```

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];
```

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];
```

PADDQ (Legacy SSE instruction)

```
DEST[63:0]← DEST[63:0]  + SRC[63:0];
DEST[127:64]← DEST[127:64] + SRC[127:64];
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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[MAX_VL-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];
```

VPADDD (EVEX encoded versions)

```
(KL, VL) = (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;
```

VPADDQ (EVEX encoded versions)

```
(KL, VL) = (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;
```

Intel C/C++ Compiler Intrinsic Equivalents

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);
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);
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)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PAND—Logical AND

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F DB /r PAND xmm1, xmm2/m128	RM	V/V	SSE2	Bitwise AND of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG DB /r VPAND xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Bitwise AND of xmm2, and xmm3/m128 and store result in xmm1.
VEX.NDS.256.66.0F.WIG DB /r VPAND ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Bitwise AND of ymm2, and ymm3/m256 and store result in ymm1.
EVEX.NDS.512.66.0F.W0 DB /r VPANDD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst)	FV	V/V	AVX512F	Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.NDS.512.66.0F.W1 DB /r VPANDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst)	FV	V/V	AVX512F	Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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.

EVEX.512 encoded version: The first source operand is a ZMM 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 ZMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 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 (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

PAND (Legacy SSE instruction)

```
DEST[127:0] ← (DEST[127:0] BITWISE AND SRC[127:0])
```

VPAND (VEX.128 encoded instruction)

```
DEST[127:0] ← (SRC1[127:0] AND SRC2[127:0])
DEST[MAX_VL-1:128] ← 0
```

VPAND (VEX.256 encoded instruction)

```
DEST[255:0] ← (SRC1[255:0] AND SRC2[255:0])
DEST[MAX_VL-1:256] ← 0
```

VPANDD (EVEX encoded versions)

```
(KL, VL) = (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
```

VPANDQ (EVEX encoded versions)

```
(KL, VL) = (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
```

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);
PAND __m128i _mm_and_si128 (__m128i a, __m128i b)
VPAND __m256i _mm256_and_si256 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PANDN—Logical AND NOT

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F DF /r PANDN xmm1, xmm2/m128	RM	V/V	SSE2	Bitwise AND NOT of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG DF VPANDN xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Bitwise AND NOT of xmm2, and xmm3/m128 and store result in xmm1.
VEX.NDS.256.66.0F.WIG DF VPANDN ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Bitwise AND NOT of ymm2, and ymm3/m256 and store result in ymm1.
EVEX.NDS.512.66.0F.W0 DF VPANDND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Bitwise AND NOT of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.NDS.512.66.0F.W1 DF VPANDNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Bitwise AND NOT of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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.

EVEX.512 encoded version: The first source operand is a ZMM 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 ZMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 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 (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

PANDN (Legacy SSE instruction)

DEST[127:0] ← ((NOT DEST[127:0]) AND SRC[127:0])

VPANDN (VEX.128 encoded instruction)

DEST[127:0] ← ((NOT SRC1[127:0]) AND SRC2[127:0])
DEST[MAX_VL-1:128] ← 0

VPANDN (VEX.256 encoded instruction)

```
DEST[255:0] ← ((NOT SRC1[255:0]) AND SRC2[255:0])
DEST[MAX_VL-1:256] ← 0
```

VPANDND (EVEX encoded versions)

```
(KL, VL) = (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
```

VPANDNQ (EVEX encoded versions)

```
(KL, VL) = (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
```

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);
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);
PANDN __m128i _mm_andnot_si128 (__m128i a, __m128i b)
VPANDN __m256i _mm256_andnot_si256 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PCMPEQB/PCMPEQW/PCMPEQD/PCMPEQQ—Compare Packed Integers for Equality

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 74 /r PCMPEQB xmm1, xmm2/m128	RM	V/V	SSE2	Compare packed bytes in xmm2/m128 and xmm1 for equality.
66 0F 75 /r PCMPEQW xmm1, xmm2/m128	RM	V/V	SSE2	Compare packed words in xmm2/m128 and xmm1 for equality.
66 0F 76 /r PCMPEQD xmm1, xmm2/m128	RM	V/V	SSE2	Compare packed doublewords in xmm2/m128 and xmm1 for equality.
66 0F 38 29 /r PCMPEQQ xmm1, xmm2/m128	RM	V/V	SSE4_1	Compare packed quadwords in xmm2/m128 and xmm1 for equality.
VEX.NDS.128.66.0F.WIG 74 /r VPCMPEQB xmm1, xmm2, xmm3 /m128	RVM	V/V	AVX	Compare packed bytes in xmm3/m128 and xmm2 for equality.
VEX.NDS.128.66.0F.WIG 75 /r VPCMPEQW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed words in xmm3/m128 and xmm2 for equality.
VEX.NDS.128.66.0F.WIG 76 /r VPCMPEQD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed doublewords in xmm3/m128 and xmm2 for equality.
VEX.NDS.128.66.0F38.WIG 29 /r VPCMPEQQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed quadwords in xmm3/m128 and xmm2 for equality.
VEX.NDS.256.66.0F.WIG 74 /r VPCMPEQB ymm1, ymm2, ymm3 /m256	RVM	V/V	AVX2	Compare packed bytes in ymm3/m256 and ymm2 for equality.
VEX.NDS.256.66.0F.WIG 75 /r VPCMPEQW ymm1, ymm2, ymm3 /m256	RVM	V/V	AVX2	Compare packed words in ymm3/m256 and ymm2 for equality.
VEX.NDS.256.66.0F.WIG 76 /r VPCMPEQD ymm1, ymm2, ymm3 /m256	RVM	V/V	AVX2	Compare packed doublewords in ymm3/m256 and ymm2 for equality.
VEX.NDS.256.66.0F38.WIG 29 /r VPCMPEQQ ymm1, ymm2, ymm3 /m256	RVM	V/V	AVX2	Compare packed quadwords in ymm3/m256 and ymm2 for equality.
EVEX.NDS.512.66.0F.W0 76 /r VPCMPEQD k1 {k2}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Compare Equal between int32 vectors in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask k2,
EVEX.NDS.512.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare for equality of the packed bytes, words, doublewords, or quadwords in the first source operand and the second source 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 PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the PCMPEQW instruction compares the corresponding words in the destination and source operands; the PCMPEQD instruction compares the corresponding doublewords in the destination and source operands, and the PCMPEQQ instruction compares the corresponding quadwords in the destination and source operands.

Legacy SSE instructions: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. In 64-bit mode, 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 second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAX_VL-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 (MAX_VL-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 versions: The first source operand (second operand) is a vector register. The second source operand can be a vector register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

```
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;
```

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;
```

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])
```

VPCMPEQB (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAX_VL-1:128] ←0
```

PCMPEQB (128-bit Legacy SSE version)

```
DEST[127:0] ←COMPARE_BYTES_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

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])
```

VPCMPEQW (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAX_VL-1:128] ←0
```

PCMPEQW (128-bit Legacy SSE version)

```
DEST[127:0] ←COMPARE_WORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

VPCMPEQD (EVEX encoded versions)

```
(KL, VL) = (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
```

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])

VPCMPEQD (VEX.128 encoded version)
DEST[127:0] ←COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAX_VL-1:128] ←0

PCMPEQD (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_DWORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)

VPCMPEQQ (EVEX encoded versions)
(KL, VL) = (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

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])

VPCMPEQQ (VEX.128 encoded version)
DEST[127:0] ←COMPARE_QWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAX_VL-1:128] ←0

PCMPEQQ (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_QWORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPEQD __mmask16 _mm512_cmpeq_epi32_mask(__m512i a, __m512i b);
VPCMPEQD __mmask16 _mm512_mask_cmpeq_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPEQQ __mmask8 _mm512_cmpeq_epi64_mask(__m512i a, __m512i b);
VPCMPEQQ __mmask8 _mm512_mask_cmpeq_epi64_mask(__mmask8 k, __m512i a, __m512i b);
PCMPEQB __m128i _mm_cmpeq_epi8 (__m128i a, __m128i b)
PCMPEQW __m128i _mm_cmpeq_epi16 (__m128i a, __m128i b)
PCMPEQD __m128i _mm_cmpeq_epi32 (__m128i a, __m128i b)
PCMPEQQ __m128i _mm_cmpeq_epi64(__m128i a, __m128i b);
PCMPEQB __m256i _mm256_cmpeq_epi8 (__m256i a, __m256i b)
PCMPEQW __m256i _mm256_cmpeq_epi16 (__m256i a, __m256i b)
PCMPEQD __m256i _mm256_cmpeq_epi32 (__m256i a, __m256i b)

PCMPEQQ __m256i _mm256_cmpeq_epi64(__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

PCMPGTB/PCMPGTW/PCMPGTD/PCMPGTQ—Compare Packed Integers for Greater Than

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 64 /r PCMPGTB xmm1, xmm2/m128	RM	V/V	SSE2	Compare packed signed byte integers in xmm1 and xmm2/m128 for greater than.
66 0F 65 /r PCMPGTW xmm1, xmm2/m128	RM	V/V	SSE2	Compare packed signed word integers in xmm1 and xmm2/m128 for greater than.
66 0F 66 /r PCMPGTD xmm1, xmm2/m128	RM	V/V	SSE2	Compare packed signed doubleword integers in xmm1 and xmm2/m128 for greater than.
66 0F 38 37 /r PCMPGTQ xmm1, xmm2/m128	RM	V/V	SSE4_2	Compare packed qwords in xmm2/m128 and xmm1 for greater than.
VEX.NDS.128.66.0F.WIG 64 /r VPCMPGTB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than.
VEX.NDS.128.66.0F.WIG 65 /r VPCMPGTW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed word integers in xmm2 and xmm3/m128 for greater than.
VEX.NDS.128.66.0F.WIG 66 /r VPCMPGTD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed doubleword integers in xmm2 and xmm3/m128 for greater than.
VEX.NDS.128.66.0F38.WIG 37 /r VPCMPGTQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed qwords in xmm2 and xmm3/m128 for greater than.
VEX.NDS.256.66.0F.WIG 64 /r VPCMPGTB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 for greater than.
VEX.NDS.256.66.0F.WIG 65 /r VPCMPGTW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed word integers in ymm2 and ymm3/m256 for greater than.
VEX.NDS.256.66.0F.WIG 66 /r VPCMPGTD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed doubleword integers in ymm2 and ymm3/m256 for greater than.
VEX.NDS.256.66.0F38.WIG 37 /r VPCMPGTQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed qwords in ymm2 and ymm3/m256 for greater than.
EVEX.NDS.512.66.0F.W0 66 /r VPCMPGTD k1 {k2}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Compare Greater between int32 elements in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask. k2.
EVEX.NDS.512.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD signed compare for the greater value of the packed byte, word, doubleword, or quadword integers in the first source operand and the second source operand. If a data element in the first source operand is greater than the corresponding date element in the second 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 first and second source operands; the PCMPGTW instruction compares the corresponding signed word integers in the first and second source operands; the PCMPGTD instruction compares the corresponding signed doubleword integers in the first and second source operands, and the PCMPGTQ instruction compares the corresponding signed qword integers in the first and second source operands.

Legacy SSE instructions: In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15). 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.

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 (MAX_VL-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 (MAX_VL-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 versions: The first source operand (second operand) is a vector register. The second source operand can be a vector register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

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;

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;

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])

VPCMPGTB (VEX.128 encoded version)
DEST[127:0] ←COMPARE_BYTES_GREATER(SRC1[127:0],SRC2[127:0])
DEST[MAX_VL-1:128] ←0

PCMPGTB (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_BYTES_GREATER(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)

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])

VPCMPGTW (VEX.128 encoded version)
DEST[127:0] ←COMPARE_WORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[MAX_VL-1:128] ←0

PCMPGTW (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_WORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)

VPCMPGTD (EVEX encoded versions)
(KL, VL) = (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
```

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])
```

VPCMPGTD (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[MAX_VL-1:128] ←0
```

PCMPGTD (128-bit Legacy SSE version)

```
DEST[127:0] ←COMPARE_DWORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

VPCMPGTQ (EVEX encoded versions)

```
(KL, VL) = (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
```

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])
```

VPCMPGTQ (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_QWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[MAX_VL-1:128] ←0
```

PCMPGTQ (128-bit Legacy SSE version)

```
DEST[127:0] ←COMPARE_QWORDS_GREATER(DEST[127:0],SRC2[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPCMPGTD __mmask16 _mm512_cmpgt_epi32_mask(__m512i a, __m512i b);
VPCMPGTD __mmask16 _mm512_mask_cmpgt_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm512_cmpgt_epi64_mask( __m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm512_mask_cmpgt_epi64_mask(__mmask8 k, __m512i a, __m512i b);
PCMPGTB __m128i _mm_cmpgt_epi8 ( __m128i a, __m128i b)
PCMPGTW __m128i _mm_cmpgt_epi16 ( __m128i a, __m128i b)
```

PCMPGTD __m128i _mm_cmpgt_epi32 (__m128i a, __m128i b)
PCMPGTQ __m128i _mm_cmpgt_epi64(__m128i a, __m128i b);
PCMPGTB __m256i _mm256_cmpgt_epi8 (__m256i a, __m256i b)
PCMPGTW __m256i _mm256_cmpgt_epi16 (__m256i a, __m256i b)
PCMPGTD __m256i _mm256_cmpgt_epi32 (__m256i a, __m256i b)
PCMPGTQ __m256i _mm256_cmpgt_epi64(__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

VPCMPD/VPCMPUD—Compare Packed Integer Values into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, zmm2, zmm3/m512/m32bcst, imm8	FV	V/V	AVX512F	Compare packed signed doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.
EVEX.NDS.512.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, zmm2, zmm3/m512/m32bcst, imm8	FV	V/V	AVX512F	Compare packed unsigned doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, 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 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-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2. Up to sixteen comparisons are performed with results written to the destination operand.

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.

Table 5-18. 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;
```

VPCMPD (EVEX encoded versions)

```
(KL, VL) = (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) = (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);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

VPCMPQ/VPCMPUQ—Compare Packed Integer Values into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, zmm2, zmm3/m512/m64bcst, imm8	FV	V/V	AVX512F	Compare packed signed quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, zmm2, zmm3/m512/m64bcst, imm8	FV	V/V	AVX512F	Compare packed unsigned quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, 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).

VPCMPPQ/VPCMPPUQ performs a comparison between pairs of signed/unsigned quadword integer values.

The first source operand (second 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 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2. Up to eight comparisons are performed with results written to the destination operand.

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.

Table 5-19. 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;
```

VPCMPQ (EVEX encoded versions)

```
(KL, VL) = (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) = (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);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

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.512.66.0F38.W0 8B /r VPCOMPRESSD zmm1/m512 {k1}{z}, zmm2	T1S	V/V	AVX512F	Compress packed doubleword integer values from zmm2 using controlmask k1 and store to zmm1/m512.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Compress (store) up to 16 doubleword integer values from the source operand (second operand) to the destination operand (first operand).The source operand is a ZMM register, the destination operand can be a ZMM register or 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 is ignored.

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) = (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) = (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
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCOMPRESSD __m512i _mm512_mask_compress_epi32(__m512i s, __mmask16 k, __m512i a);
VPCOMPRESSD __m512i _mm512_maskz_compress_epi32(__mmask16 k, __m512i a);
VPCOMPRESSD void _mm512_mask_compressstoreu_epi32(void * a, __mmask16 k, __m512i s);

SIMD Floating-Point Exceptions

None

Other Exceptions

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

VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 8B /r VPCOMPRESSQ zmm1/mV {k1}{z}, zmm2	T1S	V/V	AVX512F	Compress packed quadword integer values from zmm2 using controlmask k1 and store to zmm1/mV.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Compress (stores) up to 8 quadword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM register, the destination operand can be a ZMM register or 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 is ignored.

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) = (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) = (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
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCOMPRESSQ __m512i _mm512_mask_compress_epi64(__m512i s, __mmask8 k, __m512i a);
VPCOMPRESSQ __m512i _mm512_maskz_compress_epi64(__mmask8 k, __m512i a);
VPCOMPRESSQ void _mm512_mask_compressstoreu_epi64(void * a, __mmask8 k, __m512i s);

SIMD Floating-Point Exceptions

None

Other Exceptions

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

VPERMD—Permute Packed Doublewords/Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.W0 36 /r VPERMD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Permute doublewords in ymm3/m256 using indices in ymm2 and store the result in ymm1.
EVEX.NDS.512.66.0F38.W0 36 /r VPERMD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Permute doublewords in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Copies doublewords 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 (MAX_VL-1:256) of the corresponding ZMM 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.

Operation

VPERMD (EVEX encoded versions)

```
(KL, VL) = (16, 512)
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
```

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[MAX_VL-1:256] ← 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);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.

VPERMI2D/VPERMI2PS/VPERMI2Q/VPERMI2PD—Full 32-bit and 64-bit Permute Overwriting the Index

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.512.66.0F38.W0 76 /r VPERMI2D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Permute double-words in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 76 /r VPERMI2Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Permute quad-words in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 77 /r VPERMI2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Permute single-precision floating-point values in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 77 /r VPERMI2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Permute double-precision values on floating-point in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Permutes 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 registers, the third operand can be a ZMM register or memory location. The first operand contains input indices to select elements from the input tables in the 2nd and 3rd operands. The first operand is also the destination of the result. 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.

Bits [3:0]/[2:0] of each element in the input index vector select an element within the two source operands. Bit 4/3 selects the first source operand if 0, otherwise selects the second source operand.

If EVEX.b is set, and the second source operand is a memory location and bit 4/3 of an index element is set, the corresponding destination element is fetched from the low 32/64-bit value at the memory location.

Note that these instructions permit a 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.

Operation

VPERMI2D/VPERMI2PS (EVEX encoded versions)

```
(KL, VL) = (16, 512)
    id ← 3
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
```

VPERMI2Q/VPERMI2PD (EVEX encoded versions)

```
(KL, VL) = (8 512)
    id ← 2
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
```

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);
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);
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);
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);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4NF.

VPERMT2D/VPERMT2PS/VPERMT2Q/VPERMT2PD—Full 32-bit and 64-bit Permute Overwriting the Table

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.512.66.0F38.W0 7E /r VPERMT2D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Permute double-words in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 7E /r VPERMT2Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Permute quad-words in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 7F /r VPERMT2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Permute single-precision values on floating-point in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 7F /r VPERMT2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Permute double-precision values on floating-point in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Permutes 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 registers, the third operand can be a ZMM register or memory location. The second operand contains input indices to select elements from the 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 register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

Bits [3:0]/[2:0] of each element in the input index vector select an element within the two input tables. Bit 4/3 selects the first operand if 0, otherwise selects the third operand.

If EVEX.b is set, and the third operand is a memory location and bit 4/3 of an index element is set, the corresponding destination element is fetched from the low 32/64-bit value at the memory location.

Note that these instructions permit a 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.

Operation

VPERMT2D/VPERMT2PS (EVEX encoded versions)

```
(KL, VL) = (16, 512)
    id ← 3
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
```

VPERMT2Q/VPERMT2PD (EVEX encoded versions)

```
(KL, VL) = (8 512)
    id ← 2
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
```

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);
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);
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);
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);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4NF.

VPERMILPD—Permute Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 0D /r VPERMILPD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Permute double-precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.
VEX.NDS.256.66.0F38.W0 0D /r VPERMILPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Permute double-precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.
EVEX.NDS.512.66.0F38.W1 0D /r VPERMILPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV-RVM	V/V	AVX512F	Permute double-precision floating-point values in zmm2 using control from zmm3/m512/m64bcst and store the result in zmm1 using writemask k1.
VEX.128.66.0F3A.W0 05 /r ib VPERMILPD xmm1, xmm2/m128, imm8	RM	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	RM	V/V	AVX	Permute double-precision floating-point values in ymm2/m256 using controls from imm8.
EVEX.512.66.0F3A.W1 05 /r ib VPERMILPD zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FV-RM	V/V	AVX512F	Permute double-precision floating-point values in zmm2/m512/m64bcst using controls from imm8 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

(variable control version)

Permute double-precision floating-point values in the first source operand (second operand) using 8-bit control fields in the low bytes of the second source operand (third operand) and store results in the destination operand (first operand). The destination and the first source operand are vector register.

There is one control byte per destination double-precision element. Each control byte is aligned with the low 8 bits of the corresponding double-precision destination element. Each control byte contains a 1-bit select field (see Figure 5-30) that determines which of the source elements are selected. Source elements are restricted to lie in the same source 128-bit region as the destination.

EVEX.512 version: The second source operand (third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask.

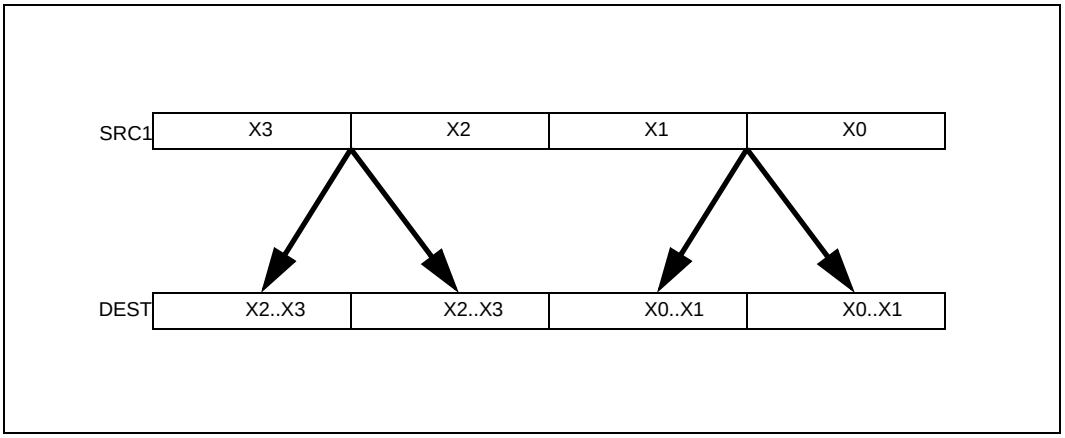


Figure 5-29. VPERMILPD Operation

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

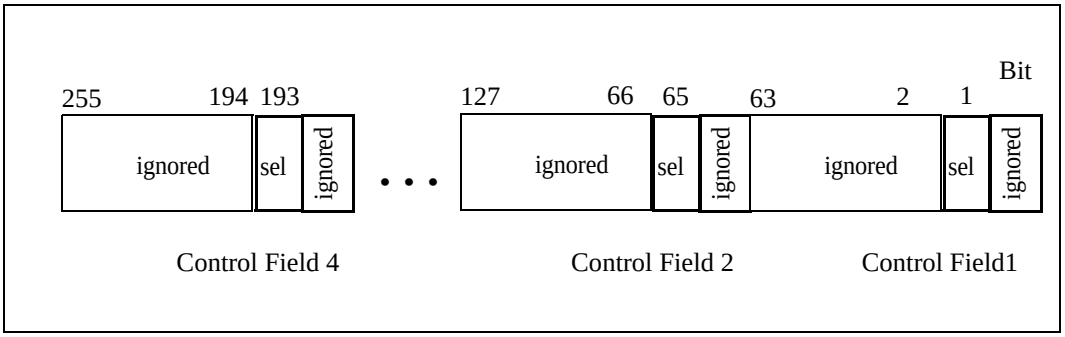


Figure 5-30. VPERMILPD Shuffle Control

(immediate control version)

Permute double-precision floating-point values in the first source operand (second operand) using, 1-bit control fields in the 8-bit immediate and store results in the destination operand (first operand), the same control bits used for the lower 256-bit are used for the upper half.

VEX version: The source operand is a vector register or a memory location and the destination operand is a vector register.

EVEX.512 version: The source operand (second operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask.

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 (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;
    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;
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
```

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]
```

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[MAX_VL-1:128]←0
```

VPERMILPD (EVEX variable versions)

```
(KL, VL) = (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 (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;
    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;

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
```

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]
```

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[MAX_VL-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 __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 __m128d _mm_permute_pd (__m128d a, int control)
VPERMILPD __m256d _mm256_permute_pd (__m256d a, int control)
VPERMILPD __m128d _mm_permutevar_pd (__m128d a, __m128i control);
VPERMILPD __m256d _mm256_permutevar_pd (__m256d a, __m256i control);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD	If VEX.W = 1.

EVEX-encoded instruction, see Exceptions Type E4NF.

VPERMILPS—Permute Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 0C /r VPERMILPS xmm1, xmm2, xmm3/m128	RVM	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	RM	V/V	AVX	Permute single-precision floating-point values in xmm2/m128 using controls from imm8 and store result in xmm1.
VEX.NDS.256.66.0F38.W0 0C /r VPERMILPS ymm1, ymm2, ymm3/m256	RVM	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	RM	V/V	AVX	Permute single-precision floating-point values in ymm2/m256 using controls from imm8 and store result in ymm1.
EVEX.NDS.512.66.0F38.W0 0C /r VPERMILPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV-RVM	V/V	AVX512F	Permute single-precision floating-point values zmm2 using control from zmm3/m512/m32bcst and store the result in zmm1 using writemask k1.
EVEX.512.66.0F3A.W0 04 /r ib VPERMILPS zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	FV-RM	V/V	AVX512F	Permute single-precision floating-point values zmm2/m512/m32bcst using controls from imm8 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

(variable control version)

Permute single-precision floating-point values in the first source operand (second operand) using 8-bit control fields in the low bytes of corresponding elements of the shuffle control (third operand) and store results in the destination operand (first operand). The destination and the first source operand are vector register.

There is one control byte per destination single-precision element. Each control byte is aligned with the low 8 bits of the corresponding single-precision destination element. Each control byte contains a 2-bit select field (see Figure 5-32) that determines which of the source elements are selected. Source elements are restricted to lie in the same source 128-bit region as the destination.

EVEX.512 version: The second source operand (third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

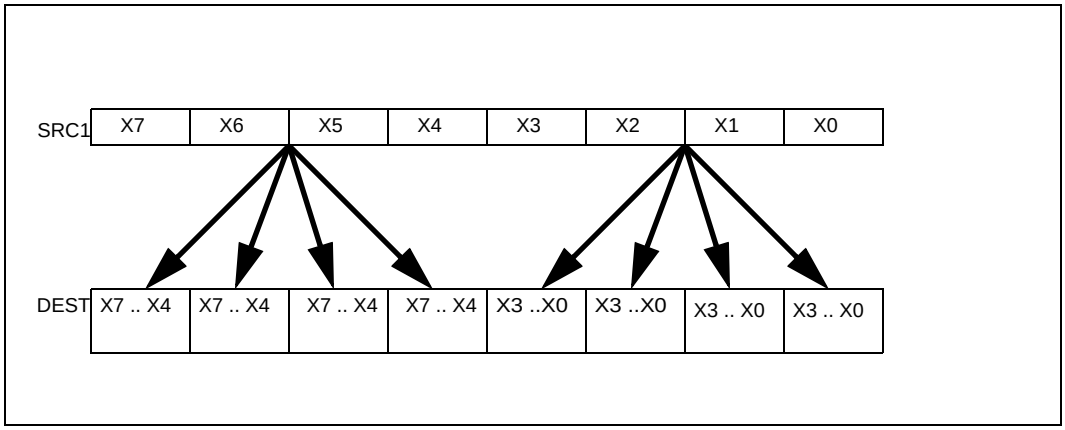


Figure 5-31. VPERMILPS Operation

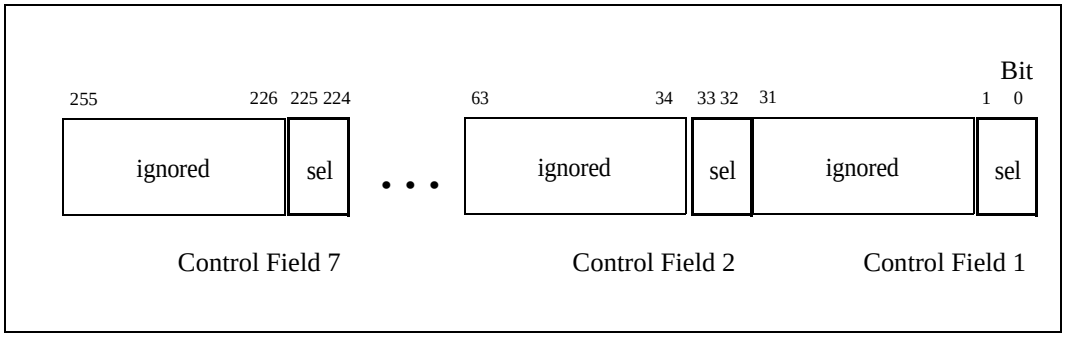


Figure 5-32. VPERMILPS Shuffle Control

(immediate control version)

Permute single-precision floating-point values in the first source operand (second operand) using four 2-bit control fields in the 8-bit immediate and store results in the destination operand (first operand); the same control bits used for the lower 256-bit are used for the upper half.

VEX version: The source operand is a vector register or a memory location and the destination operand is a vector register. This is similar to a wider version of PSHUFD, just operating on single-precision floating-point values.

EVEX.512 version: The source operand (second operand) is a ZMM register, a 512-bit memory location or a 512-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) = (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;
    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;
    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;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*
                THEN *DEST[i+31:i] remains unchanged*
                ELSE DEST[i+31:i] ← 0                    ;zeroing-masking
            FI;
    FI;
ENDFOR
```

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

VPERMILPS (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]);
```

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[MAX_VL-1:128]←0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMILPS __m512 _mm512_permute_ps(__m512d a, int imm);
VPERMILPS __m512 _mm512_mask_permute_ps(__m512 s, __mmask16 k, __m512d a, int imm);
VPERMILPS __m512 _mm512_maskz_permute_ps(__mmask16 k, __m512d 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);
VPERM1LPS __m128 _mm_permute_ps (__m128 a, int control);
VPERM1LPS __m256 _mm256_permute_ps (__m256 a, int control);
VPERM1LPS __m128 _mm_permutevar_ps (__m128 a, __m128i control);
VPERM1LPS __m256 _mm256_permutevar_ps (__m256 a, __m256i control);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD	If VEX.W = 1.

EVEX-encoded instruction, see Exceptions Type E4NF.

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	RMI	V/V	AVX2	Permute double-precision floating-point elements in ymm2/m256 using indices in imm8 and store the result in ymm1.
EVEX.512.66.0F3A.W1 01 /r /ib VPERMPD zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FV-RMI	V/V	AVX512F	Permute double-precision floating-point elements in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1 subject to writemask k1.
EVEX.NDS.512.66.0F38.W1 16 /r VPERMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV-RVM	V/V	AVX512F	Permute double-precision floating-point elements in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
FV-RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

The imm8 version: Copies quadword elements of double-precision floating-point values from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAX_VL-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.

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) = (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];
    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];
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
```

VPERMPD (EVEX - vector control forms)

```
(KL, VL) = (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;
    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];
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
```

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];
```

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);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E4NF.

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	RVM	V/V	AVX2	Permute single-precision floating-point elements in ymm3/m256 using indices in ymm2 and store the result in ymm1.
EVEX.NDS.512.66.0F38.W0 16 /r VPERMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Permute single-precision floating-point values in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 subject to write mask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Copies doubleword elements of single-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword in the source operand to be copied to more than one location in the destination operand.

VEX.256 versions: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAX_VL-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) = (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;

    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];
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
```

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];
```

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);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E4NF.

VPERMQ—Qwords Element Permutation

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 00 /r /ib VPERMQ ymm1, ymm2/m256, imm8	RMI	V/V	AVX2	Permute qwords in ymm2/m256 using indices in imm8 and store the result in ymm1.
EVEX.512.66.0F3A.W1 00 /r /ib VPERMQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FV-RMI	V/V	AVX512F	Permute qwords in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1.
EVEX.NDS.512.66.0F38.W1 36 /r VPERMQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV-RVM	V/V	AVX512F	Permute qwords in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
FV-RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

The imm8 version: Copies quadwords from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAX_VL-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.

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

Operation

VPERMQ (EVEX - imm8 control forms)

```
(KL, VL) = (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];
    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];
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
```

VPERMQ (EVEX - vector control forms)

```
(KL, VL) = (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;
    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_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];
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
```

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];
```

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);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E4NF.

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.512.66.0F38.W0 89 /r VPEXPANDD zmm1 {k1}{z}, zmm2/mV	T1S	V/V	AVX512F	Expand packed double-word integer values from zmm2/mV to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Expand (load) up to 16 contiguous doubleword integer values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

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

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

Operation

VPEXPANDD (EVEX encoded versions)

```
(KL, VL) = (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
```

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);

SIMD Floating-Point Exceptions

None

Other Exceptions

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

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.512.66.0F38.W1 89 /r VPEXPANDQ zmm1 {k1}{z}, zmm2/mV	T1S	V/V	AVX512F	Expand packed quad-word integer values from zmm2/mV to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Expand (load) up to 8 quadword integer values from the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

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

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

Operation

VPEXPANDQ (EVEX encoded versions)

```
(KL, VL) = (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
```

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);

SIMD Floating-Point Exceptions

None

Other Exceptions

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

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.512.66.0F38.W0 90 /vsib VPGATHERDD zmm1 {k1}, vm32z	T1S	V/V	AVX512F	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W1 90 /vsib VPGATHERDQ zmm1 {k1}, vm32y	T1S	V/V	AVX512F	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (r, w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

Description

A set of 16 or 8 doubleword/quadword memory locations pointed to by base address BASE_ADDR and index vector VINDEX with scale SCALE are gathered. The result is written into vector zmm1. The elements are specified via the VSIB (i.e., the index register is a zmm, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register (zmm1) is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

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 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, 2 or 4 byte displacement

VPGATHERDD (EVEX encoded version)

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        THEN DEST[i+31:i] ← MEM[BASE_ADDR +
                        SignExtend(VINDEX[i+31:i]) * SCALE + DISP]), 1)
             k1[j] ← 0
        ELSE *DEST[i+31:i] ← remains unchanged*          ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
```

VPGATHERDQ (EVEX encoded version)

```
(KL, VL) = (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
```

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);
VPGATHERDQ __m512i _mm512_i32gather_epi64(__m256i vdx, void * base, int scale);
VPGATHERDQ __m512i _mm512_mask_i32gather_epi64(__m512i s, __mmask8 k, __m256i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12.

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.512.66.0F38.W0 91 /vsib VPGATHERQD ymm1 {k1}, vm64z	T1S	V/V	AVX512F	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W1 91 /vsib VPGATHERQQ zmm1 {k1}, vm64z	T1S	V/V	AVX512F	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (r, w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

Description

A set of 8 doubleword/quadword memory locations pointed to by base address BASE_ADDR and index vector VINDEX with scale SCALE are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

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 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, 2 or 4 byte displacement

VPGATHERQD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j]
        THEN DEST[i+31:i] ← MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP]), 1)
            k1[j] ← 0
        ELSE *DEST[i+31:i] ← remains unchanged*          ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
DEST[MAX_VL-1:VL/2] ← 0
```

VPGATHERQQ (EVEX encoded version)

```
(KL, VL) = (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
```

Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERQD __m256i _mm512_i64gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERQD __m256i _mm512_mask_i64gather_epi32(__m256i s, __mmask8 k, __m512i 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);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12.

PMAXSB/PMAXSW/PMAXSD/PMAXSQ—Maximum of Packed Signed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3C /r PMAXSB xmm1, xmm2/m128	RM	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	RM	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	RM	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3C /r VPMAXSB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F.WIG EE /r VPMAXSW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3D /r VPMAXSD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.256.66.0F38.WIG 3C /r VPMAXSB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.NDS.256.66.0F.WIG EE /r VPMAXSW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and store packed maximum values in ymm1.
VEX.NDS.256.66.0F38.WIG 3D /r VPMAXSD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.NDS.512.66.0F38.W0 3D /r VPMAXSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 3D /r VPMAXSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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.

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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a vector register; The second 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 conditionally updated based on writemask k1.

Operation

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[MAX_VL-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[MAX_VL-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;
```

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[MAX_VL-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[MAX_VL-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;
```

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[MAX_VL-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[MAX_VL-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;
```

VPMAXSD (EVEX encoded versions)

```
(KL, VL) = (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
```

VPMAXSQ (EVEX encoded versions)

```
(KL, VL) = (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;
```

Intel C/C++ Compiler Intrinsic Equivalent

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);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

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	RM	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3F /r VPMAXUD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.256.66.0F38.WIG 3F /r VPMAXUD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.NDS.512.66.0F38.W0 3F /r VPMAXUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 under writemask k1.
EVEX.NDS.512.66.0F38.W1 3F /r VPMAXUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a vector register; The second 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 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[MAX_VL-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[MAX_VL-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[MAX_VL-1:256] ← 0
```

VPMAXUD (EVEX encoded versions)

```
(KL, VL) = (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;
```

VPMAXUQ (EVEX encoded versions)

```
(KL, VL) = (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;
```

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);
(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 Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

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	RM	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 39 /r VPMINSD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.256.66.0F38.WIG 39 /r VPMINSD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m128 and store packed minimum values in ymm1.
EVEX.NDS.512.66.0F38.W0 39 /r VPMINSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.NDS.512.66.0F38.W1 39 /r VPMINSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a vector register; The second 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 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[MAX_VL-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[MAX_VL-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[MAX_VL-1:256] ← 0
```

VPMINSD (EVEX encoded versions)

```
(KL, VL) = (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;
```

VPMINSQ (EVEX encoded versions)

```
(KL, VL) = (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;
```

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);
(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 Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

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	RM	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3B /r VPMINUD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.256.66.0F38.WIG 3B /r VPMINUD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
EVEX.NDS.512.66.0F38.W0 3B /r VPMINUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.NDS.512.66.0F38.W1 3B /r VPMINUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed unsigned 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.

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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a vector register; The second 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 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[MAX_VL-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[MAX_VL-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[MAX_VL-1:256] ← 0
```

VPMINUD (EVEX encoded versions)

```
(KL, VL) = (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;
```

VPMINUQ (EVEX encoded versions)

```
(KL, VL) = (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;
```

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);
(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 Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

VPMOVQB/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.F3.0F38.W0 32 /r VPMOVQB *xmm1/mV {k1}{z}, zmm2*	OVM	V/V	AVX512F	Converts 8 packed quad-word integers from *zmm2* into 8 packed byte integers in *xmm1/mV* with truncation under writemask k1.
EVEX.512.F3.0F38.W0 22 /r VPMOVSQB *xmm1/mV {k1}{z}, zmm2*	OVM	V/V	AVX512F	Converts 8 packed signed quad-word integers from *zmm2* into 8 packed signed byte integers in *xmm1/mV* using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 12 /r VPMOVUSQB *xmm1/mV {k1}{z}, zmm2*	OVM	V/V	AVX512F	Converts 8 packed unsigned quad-word integers from *zmm2* into 8 packed unsigned byte integers in *xmm1/mV* using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
OVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

VPMOVQB down converts 64-bit integer elements in the source operand (the second operand) into packed byte elements using truncation. VPMOVSQB converts signed 64-bit integers into packed signed bytes using signed saturation. VPMOVUSQB convert unsigned quad-word values into unsigned byte values using unsigned saturation. The source operand is a vector register. The destination operand is an XMM register or a memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAX_VL-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) = (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[MAX_VL-1:VL/8] ← 0;
```

VPMOVQB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (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) = (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[MAX_VL-1:VL/8] ← 0;
```

VPMOVSQB instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (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) = (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[MAX_VL-1:VL/8] ← 0;
```

VPMOVUSQB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (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);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6NF.

VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.F3.0F38.W0 34 /r VPMOVQW *xmm1/mV {k1}{z}, zmm2*	QVM	V/V	AVX512F	Converts 8 packed quad-word integers from *zmm2* into 8 packed word integers in *xmm1/mV* with truncation under writemask k1.
EVEX.512.F3.0F38.W0 24 /r VPMOVSQW *xmm1/mV {k1}{z}, zmm2*	QVM	V/V	AVX512F	Converts 8 packed signed quad-word integers from *zmm2* into 8 packed signed word integers in *xmm1/mV* using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 14 /r VPMOVUSQW *xmm1/mV {k1}{z}, zmm2*	QVM	V/V	AVX512F	Converts 8 packed unsigned quad-word integers from *zmm2* into 8 packed unsigned word integers in *xmm1/mV* using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
QVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed words using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned word values using unsigned saturation.

The source operand is a vector register. The destination operand is an XMM register or a 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 (MAX_VL-1:128) of the 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) = (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[MAX_VL-1:VL/4] ← 0;
```

VPMOVQW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (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) = (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[MAX_VL-1:VL/4] ← 0;
```

VPMOVSQW instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (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) = (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[MAX_VL-1:VL/4] ← 0;
```

VPMOVUSQW instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (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);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6NF.

VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.F3.0F38.W0 35 /r VPMOVQD *ymm1/mV {k1}{z}, zmm2*	HVM	V/V	AVX512F	Converts 8 packed quad-word integers from *zmm2* into 8 packed double-word integers in *ymm1/mV* with truncation subject to writemask k1.
EVEX.512.F3.0F38.W0 25 /r VPMOVSQD *ymm1/mV {k1}{z}, zmm2*	HVM	V/V	AVX512F	Converts 8 packed signed quad-word integers from *zmm2* into 8 packed signed double-word integers in *ymm1/mV* using signed saturation subject to writemask k1.
EVEX.512.F3.0F38.W0 15 /r VPMOVUSQD *ymm1/mV {k1}{z}, zmm2*	HVM	V/V	AVX512F	Converts 8 packed unsigned quad-word integers from *zmm2* into 8 packed unsigned double-word integers in *ymm1/mV* using unsigned saturation subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
HVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed double-words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed doublewords using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned double-word values using unsigned saturation.

The source operand is a vector register. The destination operand is a vector register or a 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 (MAX_VL-1:256) of the 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) = (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[MAX_VL-1:VL/2] ← 0;
```

VPMOVQD instruction (EVEX encoded version) memory form

```
(KL, VL) = (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) = (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[MAX_VL-1:VL/2] ← 0;
```

VPMOVSQD instruction (EVEX encoded version) memory form

```
    (KL, VL) = (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) = (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[MAX_VL-1:VL/2] ← 0;
```

VPMOVUSQD instruction (EVEX encoded version) memory form

```
    (KL, VL) = (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);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6NF.

VPMOVDB/VPMOVSDB/VPMOVUSDB—Down Convert DWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.F3.0F38.W0 31 /r VPMOVDB *xmm1/mV {k1}{z}, zmm2*	QVM	V/V	AVX512F	Converts 16 packed double-word integers from *zmm2* into 16 packed byte integers in *xmm1/mV* with truncation under writemask k1.
EVEX.512.F3.0F38.W0 21 /r VPMOVSDB *xmm1/mV {k1}{z}, zmm2*	QVM	V/V	AVX512F	Converts 16 packed signed double-word integers from *zmm2* into 16 packed signed byte integers in *xmm1/mV* using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 11 /r VPMOVUSDB *xmm1/mV {k1}{z}, zmm2*	QVM	V/V	AVX512F	Converts 16 packed unsigned double-word integers from *zmm2* into 16 packed unsigned byte integers in *xmm1/mV* using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
QVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

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 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 (MAX_VL-1:128) of the 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) = (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[MAX_VL-1:VL/4] ← 0;
```

VPMOVDB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (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) = (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[MAX_VL-1:VL/4] ← 0;
```

VPMOVSDB instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (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) = (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[MAX_VL-1:VL/4] ← 0;
```

VPMOVUSDB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (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);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6NF.

VPMOVDW/VPMOVSDW/VPMOVUSDW—Down Convert DWord to Word

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.F3.0F38.W0 33 /r VPMOVDW *ymm1/mV {k1}{z}, zmm2*	HVM	V/V	AVX512F	Converts 16 packed double-word integers from *zmm2* into 16 packed word integers in *ymm1/mV* with truncation under writemask k1.
EVEX.512.F3.0F38.W0 23 /r VPMOVSDW *ymm1/mV {k1}{z}, zmm2*	HVM	V/V	AVX512F	Converts 16 packed signed double-word integers from *zmm2* into 16 packed signed word integers in *ymm1/mV* using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 13 /r VPMOVUSDW *ymm1/mV {k1}{z}, zmm2*	HVM	V/V	AVX512F	Converts 16 packed unsigned double-word integers from *zmm2* into 16 packed unsigned word integers in *ymm1/mV6* using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
HVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

VPMOVDW down converts 32-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSDW converts signed 32-bit integers into packed signed words using signed saturation. VPMOVUSDW convert unsigned double-word values into unsigned word values using unsigned saturation.

The source operand is a vector register. The destination operand is a vector register or a 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 (MAX_VL-1:256) of the 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) = (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[MAX_VL-1:VL/2] ← 0;
```

VPMOVDW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (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) = (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[MAX_VL-1:VL/2] ← 0;
```

VPMOVSDW instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (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) = (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[MAX_VL-1:VL/2] ← 0;
```

VPMOVUSDW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (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);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6NF.

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	RM	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	RM	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	RM	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	RM	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/m3	RM	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	RM	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	RM	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	RM	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	RM	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	RM	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	RM	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	RM	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	RM	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	RM	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	RM	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	RM	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.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1, xmm2/m64	RM	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	RM	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.
EVEX.512.66.0F38.WIG 21 /r VPMOVSXBD zmm1 {k1}{z}, xmm2/mV	QVM	V/V	AVX512F	Sign extend 16 packed 8-bit integers in the low 16 bytes of xmm2/mV to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 22 /r VPMOVSXBQ zmm1 {k1}{z}, xmm2/mV	OVM	V/V	AVX512F	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/mV to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 23 /r VPMOVSXWD zmm1 {k1}{z}, ymm2/mV	HVM	V/V	AVX512F	Sign extend 16 packed 16-bit integers in the low 32 bytes of ymm2/mem to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 24 /r VPMOVSXWQ zmm1 {k1}{z}, xmm2/mV	QVM	V/V	AVX512F	Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/mV to 8 packed 64-bit integers in zmm1 subject to writemask k1
EVEX.512.66.0F38.W0 25 /r VPMOVSXDQ zmm1 {k1}{z}, ymm2/mV	HVM	V/V	AVX512F	Sign extend 8 packed 32-bit integers in the low 32 bytes of ymm2/mV to 8 packed 64-bit integers in zmm1 using writemask k1

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
HVM, QVM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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 (MAX_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAX_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: Bits (MAX_VL-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 sign extended to quadword integers and stored to the destination operand under the writemask. The destination register is 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]);
```

VPMOVSXBD (EVEX encoded versions)

```
(KL, VL) = (16, 512)
Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
    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])
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
```

VPMOVSXBQ (EVEX encoded versions)

```
(KL, VL) = (8, 512)
Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
    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])
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
```

VPMOVSXWD (EVEX encoded versions)

```
(KL, VL) = (16, 512)
Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
    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])
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
```

VPMOVSXWQ (EVEX encoded versions)

```
(KL, VL) = (8, 512)
Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
    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])
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
```

VPMOVSXDQ (EVEX encoded versions)

```
(KL, VL) = (8, 512)
```

```
Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
    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])
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
```

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])
```

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])
```

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])
```

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])
```

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])
```

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])
```

VPMOVSXBW (VEX.128 encoded version)

```
Packed_Sign_Extend_BYTE_to_WORDDEST[127:0], SRC[127:0]()
DEST[MAX_VL-1:128] ←0
```

VPMOVSXBD (VEX.128 encoded version)

```
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAX_VL-1:128] ←0
```

VPMOVSXBQ (VEX.128 encoded version)

```
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAX_VL-1:128] ←0
```

VPMOVSXWD (VEX.128 encoded version)
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAX_VL-1:128] ←0

VPMOVSXWQ (VEX.128 encoded version)
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAX_VL-1:128] ←0

VPMOVSXDQ (VEX.128 encoded version)
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAX_VL-1:128] ←0

PMOVSXBW
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)

PMOVSXBD
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)

PMOVSXBQ
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)

PMOVSXWD
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)

PMOVSXWQ
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)

PMOVSXDQ
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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);
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 Exceptions Type 5.
EVEX-encoded instruction, see Exceptions Type E5.

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	RM	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	RM	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	RM	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	RM	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	RM	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	RM	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	RM	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	RM	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	RM	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	RM	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	RM	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	RM	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	RM	V/V	AVX2	Zero extend 16 packed 8-bit integers in the low 16 bytes of xmm2/m128 to 16 packed 16-bit integers in ymm1.
VEX.256.66.0F38.WIG 31 /r VPMOVZXBD ymm1, xmm2/m64	RM	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	RM	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	RM	V/V	AVX2	Zero extend 8 packed 16-bit integers in the low 16 bytes of 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	RM	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	RM	V/V	AVX2	Zero extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in ymm1.
EVEX.512.66.0F38.WIG 31 /r VPMOVZXBD zmm1 {k1}{z}, xmm2/mV	QVM	V/V	AVX512F	Zero extend 16 packed 8-bit integers in the low 16 bytes of xmm2/mV to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 32 /r VPMOVZXBQ zmm1 {k1}{z}, xmm2/mV	OVM	V/V	AVX512F	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/mV to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 33 /r VPMOVZXWD zmm1 {k1}{z}, ymm2/mV	HVM	V/V	AVX512F	Zero extend 16 packed 16-bit integers in the low 32 bytes of ymm2/mV to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 34 /r VPMOVZXWQ zmm1 {k1}{z}, xmm2/mV	QVM	V/V	AVX512F	Zero extend 8 packed 16-bit integers in the low 16 bytes of xmm2/mV to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W0 35 /r VPMOVZXDQ zmm1 {k1}{z}, ymm2/mV	HVM	V/V	AVX512F	Zero extend 8 packed 32-bit integers in the low 32 bytes of ymm2/V to 8 packed 64-bit integers in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
HVM, QVM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Legacy and VEX 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 (MAX_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAX_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: Bits (MAX_VL-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 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]);
```

VPMOVZXBD (EVEX encoded versions)

```
(KL, VL) = (16, 512)
Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
    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])
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
```

VPMOVZXBQ (EVEX encoded versions)

```
(KL, VL) = (8, 512)
Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
    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])
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
```

VPMOVZXWD (EVEX encoded versions)

```
(KL, VL) = (16, 512)
Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
    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
```

VPMOVZXWQ (EVEX encoded versions)

```
(KL, VL) = (8, 512)
Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
    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])
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
```

VPMOVZXDQ (EVEX encoded versions)

```
(KL, VL) = (8, 512)
Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
```

```
    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])
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
```

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])
```

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])
```

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])
```

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])
```

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])
```

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])
```

VPMOVZXBW (VEX.128 encoded version)

```
Packed_Zero_Extend_BYTE_to_WORD()
DEST[MAX_VL-1:128] ←0
```

VPMOVZXBD (VEX.128 encoded version)

```
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[MAX_VL-1:128] ←0
```

VPMOVZXBQ (VEX.128 encoded version)

```
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[MAX_VL-1:128] ←0
```

VPMOVZXWD (VEX.128 encoded version)

```
Packed_Zero_Extend_WORD_to_DWORD()
DEST[MAX_VL-1:128] ←0
```

VPMOVZXWQ (VEX.128 encoded version)
Packed_Zero_Extend_WORD_to_QWORD()
DEST[MAX_VL-1:128] ←0

VPMOVZXDQ (VEX.128 encoded version)
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[MAX_VL-1:128] ←0

PMOVZXBW
Packed_Zero_Extend_BYTE_to_WORD()
DEST[MAX_VL-1:128] (Unmodified)

PMOVZXBD
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[MAX_VL-1:128] (Unmodified)

PMOVZXBQ
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[MAX_VL-1:128] (Unmodified)

PMOVZXWD
Packed_Zero_Extend_WORD_to_DWORD()
DEST[MAX_VL-1:128] (Unmodified)

PMOVZXWQ
Packed_Zero_Extend_WORD_to_QWORD()
DEST[MAX_VL-1:128] (Unmodified)

PMOVZXDQ
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[MAX_VL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VPMOVZXBD __m512i _mm512_cvtepu8_epi32(__m512i a);
VPMOVZXBD __m512i _mm512_mask_cvtepu8_epi32(__m512i a, __mmask16 k, __m512i b);
VPMOVZXBD __m512i _mm512_maskz_cvtepu8_epi32(__mmask16 k, __m512i b);
VPMOVZXBQ __m512i _mm512_cvtepu8_epi64(__m512i a);
VPMOVZXBQ __m512i _mm512_mask_cvtepu8_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVZXBQ __m512i _mm512_maskz_cvtepu8_epi64(__mmask8 k, __m512i a);
VPMOVZXDQ __m512i _mm512_cvtepu32_epi64(__m512i a);
VPMOVZXDQ __m512i _mm512_mask_cvtepu32_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVZXDQ __m512i _mm512_maskz_cvtepu32_epi64(__mmask8 k, __m512i a);
VPMOVZXWD __m512i _mm512_cvtepu16_epi32(__m512i a);
VPMOVZXWD __m512i _mm512_mask_cvtepu16_epi32(__m512i a, __mmask16 k, __m512i b);
VPMOVZXWD __m512i _mm512_maskz_cvtepu16_epi32(__mmask16 k, __m512i a);
VPMOVZXWQ __m512i _mm512_cvtepu16_epi64(__m512i a);
VPMOVZXWQ __m512i _mm512_mask_cvtepu16_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVZXWQ __m512i _mm512_maskz_cvtepu16_epi64(__mmask8 k, __m512i 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 Exceptions Type 5.
EVEX-encoded instruction, see Exceptions Type E5.

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	RM	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.NDS.128.66.0F38.WIG 28 /r VPMULDQ xmm1, xmm2, xmm3/m128	RVM	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.NDS.256.66.0F38.WIG 28 /r VPMULDQ ymm1, ymm2, ymm3/m256	RVM	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.NDS.512.66.0F38.W1 28 /r VPMULDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Multiply packed signed doubleword integers in zmm2 by packed signed doubleword integers in zmm3/m512/m32bcst, and store the quadword results in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the first source operand by the second source operand and stores the result in the destination operand.

For PMULDQ and VPMULDQ (VEX.128 encoded version), the second source operand is two packed signed doubleword integers stored in the first (low) and third doublewords of an XMM register or an 128-bit memory location. The first source operand is two packed signed doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed signed quadword integers stored in an XMM register. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.

For VPMULDQ (VEX.256 encoded version), the second source operand is four packed signed doubleword integers stored in the first (low), third, fifth and seventh doublewords of an YMM register or an 256-bit memory location. The first source operand is four packed signed doubleword integers stored in the first, third, fifth and seventh doublewords of an YMM register. The destination contains four packed signed quadword integers stored in an YMM register. 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.

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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding 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. Bits (MAX_VL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded versions: The two source operands consist each of eight packed unsigned doubleword integers stored at odd positions (first, third,...). The destination is a ZMM register, contains up to eight packed unsigned

quadword integers and updated according to the writemask at 64-bit granularity. The second source operand can be an ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location.

Operation

VPMULDQ (EVEX encoded versions)

```
(KL, VL) = (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
```

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])
```

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[MAX_VL-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[MAX_VL-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);
(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 Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PMULLD—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	RM	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.NDS.128.66.0F38.WIG 40 /r VPMULLD xmm1, xmm2, xmm3/m128	RVM	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.NDS.256.66.0F38.WIG 40 /r VPMULLD ymm1, ymm2, ymm3/m256	RVM	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.NDS.512.66.0F38.W0 40 /r VPMULLD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD signed multiply of the packed signed dword integers in the first source operand and the second source operand and stores the low 32 bits of each intermediate result in 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 (MAX_VL-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 (MAX_VL-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 (MAX_VL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded versions: 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 conditionally updated based on writemask k1.

Operation

VPMULLD (EVEX encoded versions)

```
(KL, VL) = (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
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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_mullo_epi32(__m256i a, __m256i b);
PMULLD __m128i _mm_mullo_epi32(__m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

PMULUDQ—Multiply Packed Unsigned Doubleword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F F4 /r PMULUDQ xmm1, xmm2/m128	RM	V/V	SSE4_1	Multiply packed unsigned doubleword integers in xmm1 by packed unsigned doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.NDS.128.66.0F.WIG F4 /r VPMULUDQ xmm1, xmm2, xmm3/m128	RVM	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.NDS.256.66.0F.WIG F4 /r VPMULUDQ ymm1, ymm2, ymm3/m256	RVM	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.NDS.512.66.0F.W1 F4 /r VPMULUDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Multiply packed unsigned doubleword integers in zmm2 by packed unsigned doubleword integers in zmm3/m512/m32bcst, and store the quadword results in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies packed unsigned doubleword integers in the first source operand by the packed unsigned doubleword integers in second source operand and stores packed unsigned quadword results in the destination operand.

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 an 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 (MAX_VL-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 an 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 (MAX_VL-1:128) of the corresponding 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 an 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 unsigned quadword integers stored in an YMM register. Bits (MAX_VL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX.512 encoded version: The two sources consist each of eight packed unsigned doubleword integers stored at odd positions (first, third,...). The destination is a ZMM register, contains eight packed unsigned quadword integers and updated according to the writemask at 64-bit granularity. The second source operand can be an ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location.

Operation

VPMULUDQ (EVEX encoded versions)

```
(KL, VL) = (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
```

VPMULUDQ (VEX.256 encoded version)

```
DEST[63:0] ←ZeroExtend64( SRC1[31:0] )* ZeroExtend64( SRC2[31:0])
DEST[127:64] ←ZeroExtend64( SRC1[95:64] )* ZeroExtend64( SRC2[95:64])
DEST[191:128] ←ZeroExtend64( SRC1[159:128]) * ZeroExtend64( SRC2[159:128])
DEST[255:192] ←ZeroExtend64( SRC1[223:192]) * ZeroExtend64( SRC2[223:192])
```

VPMULUDQ (VEX.128 encoded version)

```
DEST[63:0] ←ZeroExtend64( SRC1[31:0]) * ZeroExtend64( SRC2[31:0])
DEST[127:64] ←ZeroExtend64( SRC1[95:64]) * ZeroExtend64( SRC2[95:64])
DEST[MAX_VL-1:128] ←0
```

PMULUDQ (128-bit Legacy SSE version)

```
DEST[63:0] ←DEST[31:0] * SRC[31:0]
DEST[127:64] ←DEST[95:64] * SRC[95:64]
DEST[MAX_VL-1:128] (Unmodified)
```

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_mask_mul_epu32(__mmask8 k, __m512i a, __m512i b);
(V)PMULUDQ __m128i _mm_mul_epu32(__m128i a, __m128i b);
VPMULUDQ __m256i _mm256_mul_epu32(__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

POR—Bitwise Logical Or

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F EB /r POR xmm1, xmm2/m128	RM	V/V	SSE2	Bitwise OR of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG EB /r VPOR xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Bitwise OR of xmm2/m128 and xmm3.
VEX.NDS.256.66.0F.WIG EB /r VPOR ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Bitwise OR of ymm2/m256 and ymm3.
EVEX.NDS.512.66.0F.W0 EB /r VPORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Bitwise OR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.NDS.512.66.0F.W1 EB /r VPORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Bitwise OR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical OR operation on the second source operand and the first source operand and stores the result in the destination operand. Each bit of the result is set to 0 if the corresponding bits of the first and second operands are 0; otherwise, it is set to 1.

EVEX.512 encoded version: The first source operand is a ZMM 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 ZMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 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 (MAX_VL-1:256) of the corresponding register destination are zeroed.

VEX.128 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 (MAX_VL-1:128) of the corresponding register destination are zeroed.

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 (MAX_VL-1:128) of the corresponding destination register remain unchanged.

Operation

VPORD (EVEX encoded versions)

```
(KL, VL) = (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;
```

VPORQ (EVEX encoded versions)

```
(KL, VL) = (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
                *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI;
    FI;
ENDFOR;
```

VPOR (VEX.256 encoded version)

```
DEST ←SRC1 OR SRC2
```

VPOR (VEX.128 encoded version)

```
DEST[127:0] ← (SRC[127:0] OR SRC2[127:0])
DEST[MAX_VL-1:128] ←0
```

POR (128-bit Legacy SSE version)

```
DEST[127:0] ← (SRC[127:0] OR SRC2[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

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);
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);
(V)POR __m128i _mm_or_si128 (__m128i a, __m128i b)
VPOR __m256i _mm256_or_si256 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

none

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

PROLD/PROLVD/PROLQ/PROLVQ—Bit Rotate Left

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F38.W0 15 /r VPROLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV-RVM	V/V	AVX512F	Rotate left of doublewords in zmm2 by count in the corresponding elements of zmm3/m512/m32bcst. Result written to zmm1 using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /1 ib VPROLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	FV-VMI	V/V	AVX512F	Rotate left of doublewords in zmm3/m512/m32bcst by imm8. Result written to zmm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 15 /r VPROLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV-RVM	V/V	AVX512F	Rotate quadwords in zmm2 left in the corresponding element of zmm3/m512/m64bcst under writemask k1.
EVEX.NDD.512.66.0F.W1 72 /1 ib VPROLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FV-VMI	V/V	AVX512F	Rotate quadwords in zmm2/m512/m64bcst left by imm8 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV-VMI	VEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

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) = (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
```

VPROLVD (EVEX encoded versions)

```
(KL, VL) = (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
```

VPROLQ (EVEX encoded versions)

```
(KL, VL) = (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
```

VPROLVQ (EVEX encoded versions)

```
(KL, VL) = (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
```

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);
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);
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);
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);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

PRORD/PRORVD/PRORQ/PRORVQ—Bit Rotate Right

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F38.W0 14 /r VPRORVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV-RVM	V/V	AVX512F	Rotate doublewords in zmm2 right in the corresponding element of zmm3/m512/m32bcst under writemask k1.
EVEX.NDD.512.66.0F.W0 72 /0 ib VPRORD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	FV-VMI	V/V	AVX512F	Rotate doublewords in zmm2/m512/m32bcst right by imm8 using writemask k1.
EVEX.NDS.512.66.0F38.W1 14 /r VPRORVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV-RVM	V/V	AVX512F	Rotate quadwords in zmm2 right in the corresponding element of zmm3/m512/m64bcst under writemask k1.
EVEX.NDD.512.66.0F.W1 72 /0 ib VPRORQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FV-VMI	V/V	AVX512F	Rotate quadwords in zmm2/m512/m64bcst right by imm8 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV-VMI	VEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

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) = (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
```

VPRORVD (EVEX encoded versions)

```
(KL, VL) = (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
```

VPRORQ (EVEX encoded versions)

```
(KL, VL) = (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
```

VPRORVQ (EVEX encoded versions)

```
(KL, VL) = (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
```

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);
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);
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);
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);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 A0 /vsib VPSCATTERDD vm32z {k1}, (zmm1)	T1S	V/V	AVX512F	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32y {k1}, (zmm1)	T1S	V/V	AVX512F	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.512.66.0F38.W0 A1 /vsib VPSCATTERQD vm64z {k1}, (ymm1)	T1S	V/V	AVX512F	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64z {k1}, (zmm1)	T1S	V/V	AVX512F	Using signed qword indices, scatter qword values to memory using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	ModRM:reg (r)	NA	NA

Description

Stores up to 16 elements (8 elements for qword indices) in doubleword vector or 8 elements in quadword vector to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already

scattered (i.e., if the exception is triggered by an element other than the rightmost

one with its mask bit set). When this happens, the destination register and the mask

The opmask 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 after down-conversion.

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, 2 or 4 byte displacement

VPSCATTERDD (EVEX encoded versions)

```
(KL, VL)= (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)= (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)= (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)= (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 _mm512_mask_i32scatter_epi32(void * base, __mmask16 k, __m512i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm512_i32scatter_epi64(void * base, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm512_mask_i32scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m512i a, int scale);
VPSCATTERQD void _mm512_i64scatter_epi32(void * base, __m512i vdx, __m256i a, int scale);
VPSCATTERQD void _mm512_mask_i64scatter_epi32(void * base, __mmask8 k, __m512i vdx, __m256i a, int scale);
VPSCATTERQQ void _mm512_i64scatter_epi64(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERQQ void _mm512_mask_i64scatter_epi64(void * base, __mmask8 k, __m512i vdx, __m512i a, int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12.

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	RMI	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	RMI	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	RMI	V/V	AVX2	Shuffle the doublewords in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.
EVEX.512.66.0F.W0 70 /r ib VPSHUFD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	FV	V/V	AVX512F	Shuffle the doublewords in zmm2/m512/m32bcst based on the encoding in imm8 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Copies doublewords from the source operand (the second operand) and inserts them in the destination operand (the first operand) at the locations selected with the immediate operand (third operand). Figure 5-33 shows the operation of the 256-bit VPSHUFD instruction and the encoding of the order operand imm8. 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 encoding of bits 0 and 1 of the order operand (see the field encoding in Figure 5-33) determines which doubleword (from the respective 128-bit lane) of the source operand will be copied to doubleword 0 of the destination operand.

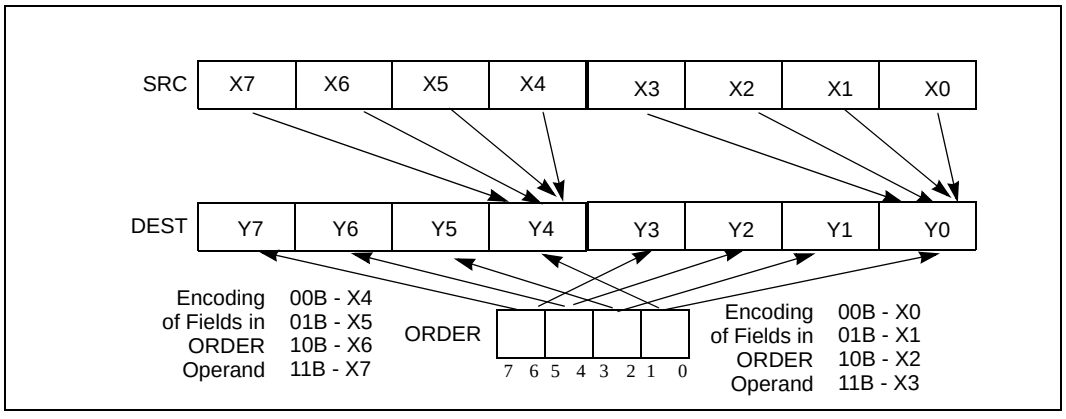


Figure 5-33. 256-bit VPSHUFD Instruction Operation

For 128-bit operation, only the low 128-bit lane are operative. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

128-bit Legacy SSE version: The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. Bits (MAX_VL-1:128) of the corresponding ZMM 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 (MAX_VL-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 (MAX_VL-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.512 encoded version: The source operand can be an 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 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

VPSHUFD (EVEX encoded versions)

```
(KL, VL) = (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;
    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];
    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];
    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];
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
```

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[MAX_VL-1:256] ← 0

VPSHUFD (VEX.128 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[MAX_VL-1:128] ← 0

PSHUFD (128-bit Legacy SSE 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[MAX_VL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFD __m512i _mm512_shuffle_epi32(__m512i a, enum);
VPSHUFD __m512i _mm512_mask_shuffle_epi32(__m512i s, __mmask16 k, __m512i a, enum);
VPSHUFD __m512i _mm512_maskz_shuffle_epi32(__mmask16 k, __m512i a, enum);
(V)PSHUFD __m128i _mm_shuffle_epi32(__m128i a, int n);
VPSHUFD __m256i _mm256_shuffle_epi32(__m256i a, int n);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.

PSLLW/PSLLD/PSLLQ—Bit Shift Left

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F F1/r PSLLW xmm1, xmm2/m128	RM	V/V	SSE2	Shift words in xmm1 left by amount specified in xmm2/m128 while shifting in 0s.
66 0F 71 /6 ib PSLLW xmm1, imm8	MI	V/V	SSE2	Shift words in xmm1 left by imm8 while shifting in 0s.
66 0F 72 /6 ib PSLLD xmm1, imm8	MI	V/V	SSE2	Shift doublewords in xmm1 left by imm8 while shifting in 0s.
66 0F 73 /6 ib PSLLQ xmm1, imm8	MI	V/V	SSE2	Shift quadwords in xmm1 left by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG F1 /r VPSLLW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW xmm1, xmm2, imm8	VMI	V/V	AVX	Shift words in xmm2 left by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG F2 /r VPSLLD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /6 ib VPSLLD xmm1, xmm2, imm8	VMI	V/V	AVX	Shift doublewords in xmm2 left by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG F3 /r VPSLLQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /6 ib VPSLLQ xmm1, xmm2, imm8	VMI	V/V	AVX	Shift quadwords in xmm2 left by imm8 while shifting in 0s.
VEX.NDS.256.66.0F.WIG F1 /r VPSLLW ymm1, ymm2, xmm3/m128	RVM	V/V	AVX2	Shift words in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /6 ib VPSLLW ymm1, ymm2, imm8	VMI	V/V	AVX2	Shift words in ymm2 left by imm8 while shifting in 0s.
VEX.NDS.256.66.0F.WIG F2 /r VPSLLD ymm1, ymm2, xmm3/m128	RVM	V/V	AVX2	Shift doublewords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /6 ib VPSLLD ymm1, ymm2, imm8	VMI	V/V	AVX2	Shift doublewords in ymm2 left by imm8 while shifting in 0s.
VEX.NDS.256.66.0F.WIG F3 /r VPSLLQ ymm1, ymm2, xmm3/m128	RVM	V/V	AVX2	Shift quadwords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /6 ib VPSLLQ ymm1, ymm2, imm8	VMI	V/V	AVX2	Shift quadwords in ymm2 left by imm8 while shifting in 0s.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F.W0 F2 /r VPSLLD zmm1 {k1}{z}, zmm2, xmm3/m128	M128	V/V	AVX512F	Shift doublewords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.NDD.512.66.0F.W0 72 /6 ib VPSLLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	FVI	V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.W1 F3 /r VPSLLQ zmm1 {k1}{z}, zmm2, xmm3/m128	M128	V/V	AVX512F	Shift quadwords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W1 73 /6 ib VPSLLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FVI	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst left by imm8 while shifting in 0s using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MI	ModRM:r/m (r, w)	Imm8	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
VMI	VEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVI	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
M128	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count 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.

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 PSLLW instruction shifts each of the words in the first source operand to the left by the number of bits specified in the count operand; the PSLLD instruction shifts each of the doublewords in the first source operand; and the PSLLQ instruction shifts the quadword (or quadwords) in the first source operand.

Legacy SSE instructions: In 64-bit mode, 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 and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. Bits (MAX_VL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX_VL-1:128) of the corresponding ZMM 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 (MAX_VL-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.

Operation

```
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 (ymm, ymm, xmm/m128) - VEX

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1, SRC2)
```

VPSLLW (ymm, imm8) - VEX

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_WORD_256b(SRC1, imm8)
```

VPSLLW (xmm, xmm, xmm/m128) - VEX

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPSLLW (xmm, imm8) - VEX

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)
```

```
DEST[MAX_VL-1:128] ←0
```

PSLLW (xmm, xmm, xmm/m128) - VEX

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

PSLLW (xmm, imm8)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)
```

VPSLLD (EVEX versions, imm8)

```
(KL, VL) = (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
```

VPSLLD (EVEX versions, xmm/m128)

```
(KL, VL) = (16, 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)

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
```

VPSLLD (ymm, ymm, xmm/m128) - VEX

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, SRC2)
```

VPSLLD (ymm, imm8) - VEX

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, imm8)
```

VPSLLD (xmm, xmm, xmm/m128) - VEX

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
```

```
DEST[MAX_VL-1:128] ←0
```

VPSLLD (xmm, imm8) - VEX

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ←0
```

PSLLD (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

PSLLD (xmm, imm8)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)
```

VPSLLQ (EVEX versions, imm8)

```
(KL, VL) = (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) = (8, 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)

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
```

VPSLLQ (ymm, ymm, xmm/m128) - VEX

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, SRC2)
```

VPSLLQ (ymm, imm8) - VEX
DEST[255:0] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, imm8)

VPSLLQ (xmm, xmm, xmm/m128) - VEX
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0

VPSLLQ (xmm, imm8) - VEX
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ←0

PSLLQ (xmm, xmm, xmm/m128)
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)

PSLLQ (xmm, imm8)
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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 __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);
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 __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);
PSLLW __m128i _mm_slli_epi16 (__m128i m, int count)
PSLLW __m128i _mm_sll_epi16 (__m128i m, __m128i count)
PSLLD __m128i _mm_slli_epi32 (__m128i m, int count)
PSLLD __m128i _mm_sll_epi32 (__m128i m, __m128i count)
PSLLQ __m128i _mm_slli_epi64 (__m128i m, int count)
PSLLQ __m128i _mm_sll_epi64 (__m128i m, __m128i count)
VPSLLW __m256i _mm256_slli_epi16 (__m256i m, int count)
VPSLLW __m256i _mm256_sll_epi16 (__m256i m, __m128i count)
VPSLLD __m256i _mm256_slli_epi32 (__m256i m, int count)
VPSLLD __m256i _mm256_sll_epi32 (__m256i m, __m128i count)
VPSLLQ __m256i _mm256_slli_epi64 (__m256i m, int count)
VPSLLQ __m256i _mm256_sll_epi64 (__m256i m, __m128i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding, see Exceptions Type 4.

Syntax with MI/VMI operand encoding, see Exceptions Type 7.

EVEX-encoded instructions:

Syntax with M128 operand encoding, see Exceptions Type E4NF.nb.

Syntax with FVI/FVM operand encoding, see Exceptions Type E4.

PSRAW/PSRAD/PSRAQ—Bit Shift Arithmetic Right

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E1/r PSRAW xmm1, xmm2/m128	RM	V/V	SSE2	Shift words in xmm1 right by amount specified in xmm2/m128 while shifting in sign bits.
66 0F 71 /4 ib PSRAW xmm1, imm8	MI	V/V	SSE2	Shift words in xmm1 right by imm8 while shifting in sign bits.
66 0F E2 /r PSRAD xmm1, xmm2/m128	RM	V/V	SSE2	Shift doublewords in xmm1 right by amount specified in xmm2/m128 while shifting in sign bits.
66 0F 72 /4 ib PSRAD xmm1, imm8	MI	V/V	SSE2	Shift doublewords in xmm1 right by imm8 while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E1 /r VPSRAW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 71 /4 ib VPSRAW xmm1, xmm2, imm8	VMI	V/V	AVX	Shift words in xmm2 right by imm8 while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E2 /r VPSRAD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 72 /4 ib VPSRAD xmm1, xmm2, imm8	VMI	V/V	AVX	Shift doublewords in xmm2 right by imm8 while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E1 /r VPSRAW ymm1, ymm2, ymm3/m128	RVM	V/V	AVX2	Shift words in ymm2 right by amount specified in ymm3/m128 while shifting in sign bits.
VEX.NDD.256.66.0F.WIG 71 /4 ib VPSRAW ymm1, ymm2, imm8	VMI	V/V	AVX2	Shift words in ymm2 right by imm8 while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E2 /r VPSRAD ymm1, ymm2, xmm3/m128	RVM	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in ymm3/m128 while shifting in sign bits.
VEX.NDD.256.66.0F.WIG 72 /4 ib VPSRAD ymm1, ymm2, imm8	VMI	V/V	AVX2	Shift doublewords in ymm2 right by imm8 while shifting in sign bits.
EVEX.NDS.512.66.0F.W0 E2 /r VPSRAD zmm1 {k1}{z}, zmm2, xmm3/m128	M128	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /4 ib VPSRAD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	FVI	V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F.W1 E2 /r VPSRAQ zmm1 {k1}{z}, zmm2, xmm3/m128	M128	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDD.512.66.0F.W1 72 /4 ib VPSRAQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FVI	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in sign bits using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MI	ModRM:r/m (r, w)	Imm8	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
VMI	VEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVI	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
M128	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count 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. 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 filled with the initial value of the sign bit.

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 first source operand to the right by the number of bits specified in the count operand.

Legacy SSE instructions: In 64-bit mode, 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 and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. Bits (MAX_VL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX_VL-1:128) of the corresponding ZMM 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 (MAX_VL-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.

Operation

```
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 ← 15;
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 ← 31;
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 ← 63;
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 ← 15;
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 ← 31;
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 (ymm, ymm, xmm/m128) - VEX

```
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
```

VPSRAW (ymm, imm8) - VEX
```
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
```

VPSRAW (xmm, xmm, xmm/m128) - VEX
```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ← 0
```

VPSRAW (xmm, imm8) - VEX
```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ← 0
```

PSRAW (xmm, xmm, xmm/m128)
```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

PSRAW (xmm, imm8)
```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)
```

VPSRAD (EVEX versions, imm8)
```
(KL, VL) = (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
```

VPSRAD (EVEX versions, xmm/m128)
```
(KL, VL) = (16, 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)

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
```

VPSRAD (ymm, ymm, xmm/m128) - VEX
```
DEST[255:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
```

VPSRAD (ymm, imm8) - VEX
```
DEST[255:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
```

VPSRAD (xmm, xmm, xmm/m128) - VEX
```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPSRAD (xmm, imm8) - VEX
```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ←0
```

PSRAD (xmm, xmm, xmm/m128)
```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

PSRAD (xmm, imm8)
```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)
```

VPSRAQ (EVEX versions, imm8)
```
(KL, VL) = (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
```

VPSRAQ (EVEX versions, xmm/m128)
```
(KL, VL) = (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
```

Intel C/C++ Compiler Intrinsic Equivalent

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 __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);
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 __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)
PSRAW __m128i _mm_srai_epi16 (__m128i m, int count)
PSRAW __m128i _mm_sra_epi16 (__m128i m, __m128i count)
VPSRAW __m256i _mm256_sra_epi16 (__m256i m, __m128i count)
PSRAD __m128i _mm_srai_epi32 (__m128i m, int count)
PSRAD __m128i _mm_sra_epi32 (__m128i m, __m128i count)
VPSRAD __m256i _mm256_sra_epi32 (__m256i m, __m128i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding, see Exceptions Type 4.

Syntax with MI/VMI operand encoding, see Exceptions Type 7.

EVEX-encoded instructions:

Syntax with M128 operand encoding, see Exceptions Type E4NF.nb.

Syntax with FVI/FVM operand encoding, see Exceptions Type E4.

PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F D1 /r PSRLW xmm1, xmm2/m128	RM	V/V	SSE2	Shift words in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.
66 0F 71 /2 ib PSRLW xmm1, imm8	MI	V/V	SSE2	Shift words in xmm1 right by imm8 while shifting in 0s.
66 0F D2 /r PSRLD xmm1, xmm2/m128	RM	V/V	SSE2	Shift doublewords in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.
66 0F 72 /2 ib PSRLD xmm1, imm8	MI	V/V	SSE2	Shift doublewords in xmm1 right by imm8 while shifting in 0s.
66 0F D3 /r PSRLQ xmm1, xmm2/m128	RM	V/V	SSE2	Shift quadwords in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.
66 0F 73 /2 ib PSRLQ xmm1, imm8	MI	V/V	SSE2	Shift quadwords in xmm1 right by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG D1 /r VPSRLW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /2 ib VPSRLW xmm1, xmm2, imm8	VMI	V/V	AVX	Shift words in xmm2 right by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG D2 /r VPSRLD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /2 ib VPSRLD xmm1, xmm2, imm8	VMI	V/V	AVX	Shift doublewords in xmm2 right by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG D3 /r VPSRLQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /2 ib VPSRLQ xmm1, xmm2, imm8	VMI	V/V	AVX	Shift quadwords in xmm2 right by imm8 while shifting in 0s.
VEX.NDS.256.66.0F.WIG D1 /r VPSRLW ymm1, ymm2, xmm3/m1288	RVM	V/V	AVX2	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /2 ib VPSRLW ymm1, ymm2, imm8	VMI	V/V	AVX2	Shift words in ymm2 right by imm8 while shifting in 0s.
VEX.NDS.256.66.0F.WIG D2 /r VPSRLD ymm1, ymm2, xmm3/m128	RVM	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /2 ib VPSRLD ymm1, ymm2, imm8	VMI	V/V	AVX2	Shift doublewords in ymm2 right by imm8 while shifting in 0s.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F.WIG D3 /r VPSRLQ ymm1, ymm2, xmm3/m128	RVM	V/V	AVX2	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /2 ib VPSRLQ ymm1, ymm2, imm8	VMI	V/V	AVX2	Shift quadwords in ymm2 right by imm8 while shifting in 0s.
EVEX.NDS.512.66.0F38.W0 D2 /r VPSRLD zmm1 {k1}{z}, zmm2, xmm3/m128	M128	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /2 ib VPSRLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	FVI	V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W1 D3 /r VPSRLQ zmm1 {k1}{z}, zmm2, xmm3/m128	M128	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W1 73 /2 ib VPSRLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FVI	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in 0s using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MI	ModRM:r/m (r, w)	Imm8	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
VMI	VEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVI	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
M128	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count 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.

The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the second source operand is a memory address, 128 bits are loaded. Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.

The PSRLW instruction shifts each of the words in the first source operand to the right by the number of bits specified in the count operand; the PSRLD instruction shifts each of the doublewords in the first source operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the first source operand.

Legacy SSE instructions: In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (MAX_VL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX_VL-1:128) of the corresponding ZMM 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 (MAX_VL-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.

Operation

```
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 (ymm, ymm, xmm/m128) - VEX

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
```

VPSRLW (ymm, imm8) - VEX

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
```

VPSRLW (xmm, xmm, xmm/m128) - VEX

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPSRLW (xmm, imm8) - VEX
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ←0
```

PSRLW (xmm, xmm, xmm/m128)
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

PSRLW (xmm, imm8)
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)
```

VPSRLD (EVEX versions, xmm/m128)
```
(KL, VL) = (16, 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)

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
```

VPSRLD (EVEX versions, imm8)
```
(KL, VL) = (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
```

VPSRLD (ymm, ymm, xmm/m128) - VEX
```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
```

VPSRLD (ymm, imm8) - VEX
```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
```

VPSRLD (xmm, xmm, xmm/m128) - VEX
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPSRLD (xmm, imm8) - VEX
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ←0
```

PSRLD (xmm, xmm, xmm/m128)
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

PSRLD (xmm, imm8)
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)
```

VPSRLQ (EVEX versions, xmm/m128)
```
(KL, VL) =(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)
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
```

VPSRLQ (EVEX versions, imm8)
```
(KL, VL) = (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
```

VPSRLQ (ymm, ymm, xmm/m128) - VEX
```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, SRC2)
```

VPSRLQ (ymm, imm8) - VEX
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, imm8)

VPSRLQ (xmm, xmm, xmm/m128) - VEX
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0

VPSRLQ (xmm, imm8) - VEX
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ←0

PSRLQ (xmm, xmm, xmm/m128)
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)

PSRLQ (xmm, imm8)
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VPSRLD __m512i _mm512_srli_epi32(__m512i a, unsigned int imm);
VPSRLD __m512i _mm512_mask_srl_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m512i _mm512_maskz_srl_epi32(__mmask16 k, __m512i 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);
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 __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);
PSRLW __m128i _mm_srli_epi16 (__m128i m, int count)
PSRLW __m128i _mm_srl_epi16 (__m128i m, __m128i count)
VPSRLW __m256i _mm256_srl_epi16 (__m256i m, __m128i count)
PSRLD __m128i _mm_srli_epi32 (__m128i m, int count)
PSRLD __m128i _mm_srl_epi32 (__m128i m, __m128i count)
VPSRLD __m256i _mm256_srl_epi32 (__m256i m, __m128i count)
PSRLQ __m128i _mm_srli_epi64 (__m128i m, int count)
PSRLQ __m128i _mm_srl_epi64 (__m128i m, __m128i count)
VPSRLQ __m256i _mm256_srl_epi64 (__m256i m, __m128i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding, see Exceptions Type 4.

Syntax with MI/VMI operand encoding, see Exceptions Type 7.

EVEX-encoded instructions:

Syntax with M128 operand encoding, see Exceptions Type E4NF.nb.

Syntax with FVI/FVM operand encoding, see Exceptions Type E4.

VPSLLVW/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 47 /r VPSLLVD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX2	Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.128.66.0F38.W1 47 /r VPSLLVQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX2	Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.256.66.0F38.W0 47 /r VPSLLVD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.NDS.256.66.0F38.W1 47 /r VPSLLVQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
EVEX.NDS.512.66.0F38.W0 47 /r VPSLLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Shift doublewords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W1 47 /r VPSLLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Shift quadwords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (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 (MAX_VL-1:128) of the corresponding ZMM 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.

EVEX encoded versions: The destination and first source operands are ZMM registers. The count operand can be either a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location The destination is conditionally updated with writemask k1.

Operation

VPSLLVD (VEX.128 version)

```
COUNT_0 ←SRC2[4 : 0]
    (* Repeat Each COUNT_i for the low 5 bits of 2nd through 4th dwords of SRC2*)
COUNT_3 ←SRC2[100 : 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[MAX_VL-1:128] ←0;
```

VPSLLVD (VEX.256 version)

```
COUNT_0 ←SRC2[4 : 0];
    (* Repeat Each COUNT_i for the low 5 bits of 2nd through 7th dwords of SRC2*)
COUNT_7 ←SRC2[228 : 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;
```

VPSLLVD (EVEX encoded version)

```
(KL, VL) = (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;
```

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[MAX_VL-1:128] ←0;
```

VPSLLVQ (VEX.256 version)

```
COUNT_0 ←SRC2[5 : 0];
    (* Repeat Each COUNT_i for the low 6 bits of 2nd through 4th dwords of SRC2*)
COUNT_3 ←SRC2[197 : 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;
```

VPSLLVQ (EVEX encoded version)

```
(KL, VL) = (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;
```

Intel C/C++ Compiler Intrinsic Equivalent

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);
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_sllv_epi32 (__m256i m, __m256i count)
VPSLLVQ __m256i _mm256_sllv_epi64 (__m256i m, __m256i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4.

VPSRLVW/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 45 /r VPSRLVD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX2	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.128.66.0F38.W1 45 /r VPSRLVQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX2	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.256.66.0F38.W0 45 /r VPSRLVD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.NDS.256.66.0F38.W1 45 /r VPSRLVQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
EVEX.NDS.512.66.0F38.W0 45 /r VPSRLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W1 45 /r VPSRLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (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 (MAX_VL-1:128) of the corresponding ZMM 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.

EVEX encoded versions: The destination and first source operands are ZMM registers. The count operand can be either a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location The destination is conditionally updated with writemask k1.

Operation

VPSRLVD (VEX.128 version)

```
COUNT_0 ←SRC2[31 : 0]
    (* Repeat Each COUNT_i for the low 5 bits of 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[MAX_VL-1:128] ←0;
```

VPSRLVD (VEX.256 version)

```
COUNT_0 ←SRC2[31 : 0];
    (* Repeat Each COUNT_i for the low 5 bits of 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[MAX_VL-1:256] ←0;
```

VPSRLVD (EVEX encoded version)

```
(KL, VL) = (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;
```

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[MAX_VL-1:128] ←0;
```

VPSRLVQ (VEX.256 version)

```
COUNT_0 ←SRC2[63 : 0];
    (* Repeat Each COUNT_i for the low 6 bits of 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[MAX_VL-1:256] ←0;
```

VPSRLVQ (EVEX encoded version)

```
(KL, VL) = (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;
```

Intel C/C++ Compiler Intrinsic Equivalent

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);
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);
VPSRLVD __m256i _mm256_srlv_epi32 (__m256i m, __m256i count)
VPSRLVQ __m256i _mm256_srlv_epi64 (__m256i m, __m256i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4.

PSUBB/PSUBW/PSUBD/PSUBQ—Packed Integer Subtract

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F F8 /r PSUBB xmm1, xmm2/m128	RM	V/V	SSE2	Subtract packed byte integers in xmm2/m128 from xmm1.
66 0F F9 /r PSUBW xmm1, xmm2/m128	RM	V/V	SSE2	Subtract packed word integers in xmm2/m128 from xmm1.
66 0F FA /r PSUBD xmm1, xmm2/m128	RM	V/V	SSE2	Subtract packed doubleword integers in xmm2/m128 from xmm1.
66 0F FB/r PSUBQ xmm1, xmm2/m128	RM	V/V	SSE2	Subtract packed quadword integers in xmm2/m128 from xmm1.
VEX.NDS.128.66.0F.WIG F8 /r VPSUBB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Subtract packed byte integers in xmm3/m128 from xmm2.
VEX.NDS.128.66.0F.WIG F9 /r VPSUBW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Subtract packed word integers in xmm3/m128 from xmm2.
VEX.NDS.128.66.0F.WIG FA /r VPSUBD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Subtract packed doubleword integers in xmm3/m128 from xmm2.
VEX.NDS.128.66.0F.WIG FB/r VPSUBQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Subtract packed quadword integers in xmm3/m128 from xmm2.
VEX.NDS.256.66.0F.WIG F8 /r VPSUBB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Subtract packed byte integers in ymm3/m256 from ymm2.
VEX.NDS.256.66.0F.WIG F9 /r VPSUBW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Subtract packed word integers in ymm3/m256 from ymm2.
VEX.NDS.256.66.0F.WIG FA /r VPSUBD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Subtract packed doubleword integers in ymm3/m256 from ymm2.
VEX.NDS.256.66.0F.WIG FB/r VPSUBQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Subtract packed quadword integers in ymm3/m256 from ymm2.
EVEX.NDS.512.66.0F.W0 FA /r VPSUBD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Subtract packed doubleword integers in zmm3/m512/m32bcst from zmm2 and store in zmm1 using writemask k1
EVEX.NDS.512.66.0F.W1 FB/r VPSUBQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Subtract packed quadword integers in zmm3/m512/m64bcst from zmm2 and store in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Subtracts the packed byte, word, doubleword, or quadword integers in the second source operand from the first source operand and stores the result in the destination operand. When a result is too large to be represented in the 8/16/32/64 integer (overflow), the result is wrapped around and the low bits are written to the destination element (that is, the carry is ignored).

Note that these instructions 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 operated on.

128-bit Legacy SSE version: 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 (MAX_VL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 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 (MAX_VL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 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 (MAX_VL-1:256) of the corresponding ZMM 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 32/64-bit memory location. The first source operand and destination operands are ZMM registers. The destination is conditionally updated with writemask k1.

Operation

VPSUBD (EVEX encoded versions)

```
(KL, VL) = (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;
```

VPSUBQ (EVEX encoded versions)

```
(KL, VL) = (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;
```

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]
```

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[MAX_VL-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[MAX_VL-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]

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[MAX_VL-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[MAX_VL-1:128] (Unmodified)

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[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]

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[MAX_VL-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[MAX_VL-1:128] (Unmodified)

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]

VPSUBQ (VEX.128 encoded version)
DEST[63:0] ←SRC1[63:0]-SRC2[63:0]
DEST[127:64] ←SRC1[127:64]-SRC2[127:64]
DEST[MAX_VL-1:128] ←0

PSUBQ (128-bit Legacy SSE version)
DEST[63:0] ←DEST[63:0]-SRC[63:0]
DEST[127:64] ←DEST[127:64]-SRC[127:64]

DEST[MAX_VL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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);
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);
PSUBB __m128i _mm_sub_epi8 (__m128i a, __m128i b)
PSUBW __m128i _mm_sub_epi16 (__m128i a, __m128i b)
PSUBD __m128i _mm_sub_epi32 (__m128i a, __m128i b)
PSUBQ __m128i _mm_sub_epi64(__m128i m1, __m128i m2)
VPSUBB __m256i _mm256_sub_epi8 (__m256i a, __m256i b)
VPSUBW __m256i _mm256_sub_epi16 (__m256i a, __m256i b)
VPSUBD __m256i _mm256_sub_epi32 (__m256i a, __m256i b)
VPSUBQ __m256i _mm256_sub_epi64(__m256i m1, __m256i m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ—Unpack High Data

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 68/r PUNPCKHBW xmm1,xmm2/m128	RM	V/V	SSE2	Interleave high-order bytes from xmm1 and xmm2/m128 into xmm1.
66 0F 69/r PUNPCKHWD xmm1,xmm2/m128	RM	V/V	SSE2	Interleave high-order words from xmm1 and xmm2/m128 into xmm1.
66 0F 6A/r PUNPCKHDQ xmm1, xmm2/m128	RM	V/V	SSE2	Interleave high-order doublewords from xmm1 and xmm2/m128 into xmm1.
66 0F 6D/r PUNPCKHQDQ xmm1, xmm2/m128	RM	V/V	SSE2	Interleave high-order quadword from xmm1 and xmm2/m128 into xmm1 register.
VEX.NDS.128.66.0F.WIG 68/r VPUNPCKHBW xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 69/r VPUNPCKHWD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Interleave high-order words from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 6A/r VPUNPCKHDQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Interleave high-order doublewords from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 6D/r VPUNPCKHQDQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Interleave high-order quadword from xmm2 and xmm3/m128 into xmm1 register.
VEX.NDS.256.66.0F.WIG 68/r VPUNPCKHBW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Interleave high-order bytes from ymm2 and ymm3/m256 into ymm1 register.
VEX.NDS.256.66.0F.WIG 69/r VPUNPCKHWD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Interleave high-order words from ymm2 and ymm3/m256 into ymm1 register.
VEX.NDS.256.66.0F.WIG 6A/r VPUNPCKHDQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Interleave high-order doublewords from ymm2 and ymm3/m256 into ymm1 register.
VEX.NDS.256.66.0F.WIG 6D/r VPUNPCKHQDQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Interleave high-order quadword from ymm2 and ymm3/m256 into ymm1 register.
EVEX.NDS.512.66.0F.W0 6A/r VPUNPCKHDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Interleave high-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register using k1 write mask.
EVEX.NDS.512.66.0F.W1 6D/r VPUNPCKHQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Interleave high-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register using k1 write mask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, and quadwords) of the first source operand and second source operand into the destination operand. (Figure 5-34 shows the unpack operation for bytes in 64-bit operands.). The low-order data elements are ignored.

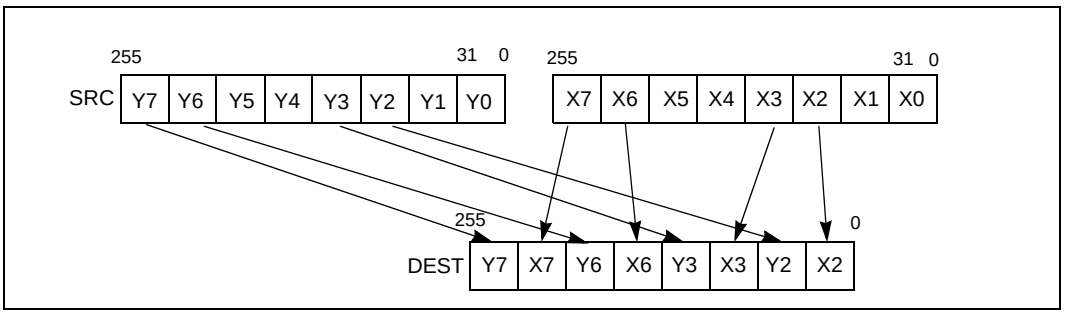


Figure 5-34. 256-bit VPUNPCKHDQ Instruction Operation

When the source data comes from a memory operand, an implementation may fetch only the appropriate half of the bits (e.g., 64 bits in 128-bit case); however, alignment rules and normal segment checking will still be enforced.

The PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the PUNPCKHDQ instruction interleaves the high order doubleword (or doublewords) of the source and destination operands, and the PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

128-bit Legacy SSE version: 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 (MAX_VL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: 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 (MAX_VL-1:128) of the corresponding 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 versions: 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 is conditionally updated with writemask k1.

Operation

```
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[511:127] ←0
```

VPUNPCKHBW (VEX.256 encoded version)

```
DEST[255:0] ←INTERLEAVE_HIGH_BYTES_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←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[511:127] ←0
```

VPUNPCKHWD (VEX.256 encoded version)

```
DEST[255:0] ←INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←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[511:127] ←0
```

VPUNPCKHDQ (VEX.256 encoded version)

```
DEST[255:0] ←INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0
```

VPUNPCKHDQ (EVEX.512 encoded version)

```
(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[VL-1:0] ← INTERLEAVE_HIGH_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])

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
```

PUNPCKHQDQ (128-bit Legacy SSE Version)

```
DEST[127:0] ←INTERLEAVE_HIGH_QWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

VPUNPCKHQDQ (VEX.128 encoded version)

```
DEST[127:0] ←INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPUNPCKHQDQ (VEX.256 encoded version)

```
DEST[255:0] ←INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0
```

VPUNPCKHQDQ (EVEX encoded versions)

```
(KL, VL) = (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;
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])

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
```

Intel C/C++ Compiler Intrinsic Equivalent

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);
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);

(V)PUNPCKHBW __m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)
VPUNPCKHBW __m256i _mm256_unpackhi_epi8(__m256i m1, __m256i m2)
(V)PUNPCKHWD __m128i _mm_unpackhi_epi16(__m128i m1,__m128i m2)
VPUNPCKHWD __m256i _mm256_unpackhi_epi16(__m256i m1,__m256i 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)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 60/r PUNPCKLBW xmm1,xmm2/m128	RM	V/V	SSE2	Interleave low-order bytes from xmm1 and xmm2/m128 into xmm1.
66 0F 61/r PUNPCKLWD xmm1,xmm2/m128	RM	V/V	SSE2	Interleave low-order words from xmm1 and xmm2/m128 into xmm1.
66 0F 62/r PUNPCKLDQ xmm1, xmm2/m128	RM	V/V	SSE2	Interleave low-order doublewords from xmm1 and xmm2/m128 into xmm1.
66 0F 6C/r PUNPCKLQDQ xmm1, xmm2/m128	RM	V/V	SSE2	Interleave low-order quadword from xmm1 and xmm2/m128 into xmm1 register.
VEX.NDS.128.66.0F.WIG 60/r VPUNPCKLBW xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 61/r VPUNPCKLWD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Interleave low-order words from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 62/r VPUNPCKLDQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Interleave low-order doublewords from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 6C/r VPUNPCKLQDQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Interleave low-order quadword from xmm2 and xmm3/m128 into xmm1 register.
VEX.NDS.256.66.0F.WIG 60/r VPUNPCKLBW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register.
VEX.NDS.256.66.0F.WIG 61/r VPUNPCKLWD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register.
VEX.NDS.256.66.0F.WIG 62/r VPUNPCKLDQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Interleave low-order doublewords from ymm2 and ymm3/m256 into ymm1 register.
VEX.NDS.256.66.0F.WIG 6C/r VPUNPCKLQDQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Interleave low-order quadword from ymm2 and ymm3/m256 into ymm1 register.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F.W0 62/r VPUNPCKLDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Interleave low-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.W1 6C/r VPUNPCKLQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the first source operand and second source operand into the destination operand. (Figure 5-35 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

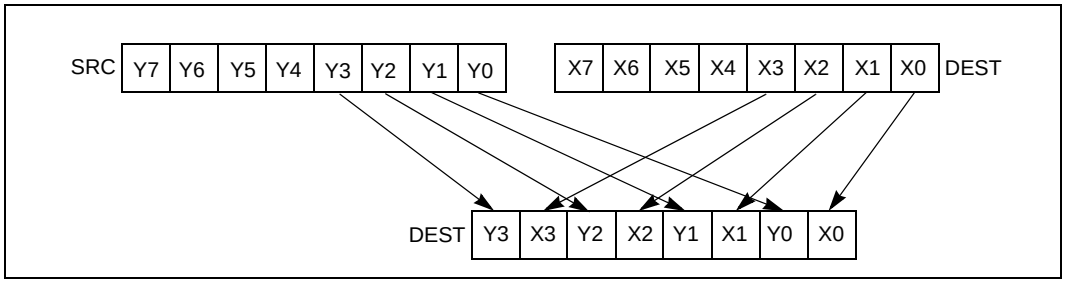


Figure 5-35. 128-bit PUNPCKLBW Instruction Operation using 64-bit Operands

When the source data comes from a memory operand, an implementation may fetch only the appropriate half of the bits (e.g., 64 bits in 128-bit case); however, alignment rules and normal segment checking will still be enforced.

The PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the PUNPCKLDQ instruction interleaves the low order doubleword (or doublewords) of the source and destination operands, and the PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

Legacy SSE instructions: In 64-bit mode, 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 second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAX_VL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: 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 (MAX_VL-1:128) of the corresponding 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 versions: The second 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 first source operand and destination operands are ZMM registers. The destination is conditionally updated with writemask k1.

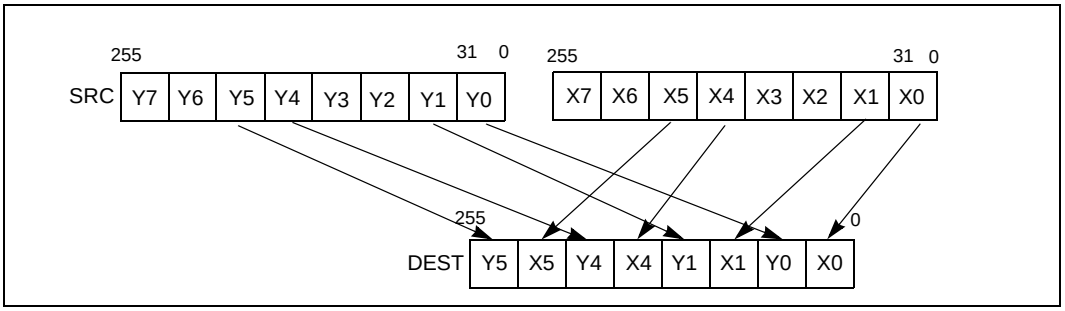


Figure 5-36. 256-bit VPUNPCKLDQ Instruction Operation

Operation

```
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] ← SRC2[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[511:127] ←0

VPUNPCKLBW (VEX.256 encoded instruction)
DEST[255:0] ←INTERLEAVE_BYTES_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0

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[511:127] ←0

VPUNPCKLWD (VEX.256 encoded instruction)
DEST[255:0] ←INTERLEAVE_WORDS_256b(SRC1, SRC2)

```
DEST[MAX_VL-1:256] ←0
```

PUNPCKLDQ

```
DEST[127:0] ←INTERLEAVE_DWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

VPUNPCKLDQ (VEX.128 encoded instruction)

```
DEST[127:0] ←INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPUNPCKLDQ (VEX.256 encoded instruction)

```
DEST[255:0] ←INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0
```

VPUNPCKLDQ (EVEX encoded instructions)

```
(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[VL-1:0] ← INTERLEAVE_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])

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)
```

PUNPCKLQDQ

```
DEST[127:0] ←INTERLEAVE_QWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

VPUNPCKLQDQ (VEX.128 encoded instruction)

```
DEST[127:0] ←INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPUNPCKLQDQ (VEX.256 encoded instruction)

```
DEST[255:0] ←INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0
```

VPUNPCKLQDQ (EVEX encoded instructions)

```
(KL, VL) = (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;
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])

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
```

Intel C/C++ Compiler Intrinsic Equivalent

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);
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);
(V)PUNPCKLBW __m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)
VPUNPCKLBW __m256i _mm256_unpacklo_epi8 (__m256i m1, __m256i m2)
(V)PUNPCKLWD __m128i _mm_unpacklo_epi16 (__m128i m1, __m128i m2)
VPUNPCKLWD __m256i _mm256_unpacklo_epi16 (__m256i m1, __m256i 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)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.

SHUFF32x4/SHUFF64x2/SHUFI32x4/SHUFI64x2—Shuffle Packed Values at 128-bit Granularity

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F3A.W0 23 /r ib VSHUFF32x4 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	FV	V/V	AVX512F	Shuffle 128-bit packed single-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.
EVEX.NDS.512.66.0F3A.W1 23 /r ib VSHUFF64x2 zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	FV	V/V	AVX512F	Shuffle 128-bit packed double-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.
EVEX.NDS.512.66.0F3A.W0 43 /r ib VSHUFI32x4 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	FV	V/V	AVX512F	Shuffle 128-bit packed double-word values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.
EVEX.NDS.512.66.0F3A.W1 43 /r ib VSHUFI64x2 zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	FV	V/V	AVX512F	Shuffle 128-bit packed quad-word values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

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.

Operation

```
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) = (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[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]);
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
```

VSHUFF64x2 (EVEX 512-bit version)

```
(KL, VL) = (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;
    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]);
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
```

VSHUFI32x4 (EVEX 512-bit version)

```
(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[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]);
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
```

VSHUFI64x2 (EVEX 512-bit version)

```
(KL, VL) = (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;
    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]);
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
```

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);
VSHUFF32x4 __m512 _mm512_shuffle_f32x4(__m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 _mm512_mask_shuffle_f32x4(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 _mm512_maskz_shuffle_f32x4(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFI64x2 __m512i _mm512_shuffle_i64x2(__m512i a, __m512i b, int imm);
VSHUFI64x2 __m512i _mm512_mask_shuffle_i64x2(__m512i s, __mmask8 k, __m512i b, __m512i b, int imm);
VSHUFI64x2 __m512i _mm512_maskz_shuffle_i64x2(__mmask8 k, __m512i a, __m512i b, int imm);

VSHUFF64x2 __m512d _mm512_shuffle_f64x2(__m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d _mm512_mask_shuffle_f64x2(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d _mm512_maskz_shuffle_f64x2(__mmask8 k, __m512d a, __m512d b, int imm);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4NF.

SHUFPD—Shuffle Packed 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	RMI	V/V	SSE2	Shuffle Packed double-precision floating-point values selected by imm8 from xmm1 and xmm2/mem.
VEX.NDS.128.66.0F.WIG C6 /r ib VSHUFPD xmm1, xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Shuffle Packed double-precision floating-point values selected by imm8 from xmm2 and xmm3/mem.
VEX.NDS.256.66.0F.WIG C6 /r ib VSHUFPD ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX	Shuffle Packed double-precision floating-point values selected by imm8 from ymm2 and ymm3/mem.
EVEX.NDS.512.66.0F.W1 C6 /r ib VSHUFPD zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	FV	V/V	AVX512F	Shuffle Packed double-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	Imm8
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

Moves either of the two packed double-precision floating-point values from each double quadword in the first source operand (second operand) into the low quadword of each double quadword of the destination operand (first operand); moves either of the two packed double-precision floating-point values from the second source operand (third operand) into the high quadword of each double quadword of the destination operand (see Figure). The immediate determines which values are moved to the destination operand.

EVEX encoded versions: The first source operand is a ZMM register. The second 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 updated according to the writemask.

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 (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

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 (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

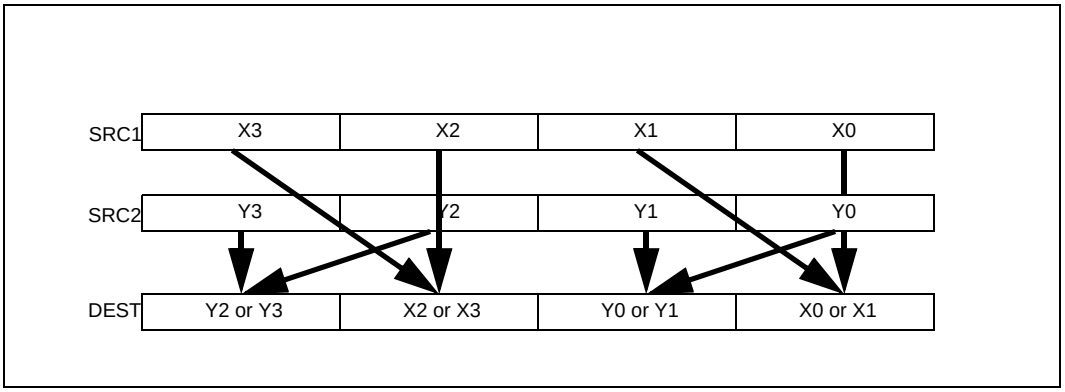


Figure 5-37. VSHUFPD Operation

Operation

VSHUFPD (EVEX encoded versions when SRC2 is a vector register)

```
(KL, VL) = (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 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;
    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;
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
```

VSHUFPD (EVEX encoded versions when SRC2 is memory)

```
(KL, VL) = (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 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;
    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;
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
```

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;
```

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[MAX_VL-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[MAX_VL-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);
SHUFPD __m128d _mm_shuffle_pd (__m128d a, __m128d b, const int select);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.

SHUFPS—Shuffle Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C6 /r ib SHUFPS xmm1, xmm3/m128, imm8	RMI	V/V	SSE	Shuffle Packed single-precision floating-point values selected by imm8 from xmm1 and xmm2/mem.
VEX.NDS.128.0F.WIG C6 /r ib VSHUFPS xmm1, xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Shuffle Packed single-precision floating-point values selected by imm8 from xmm2 and xmm3/mem.
VEX.NDS.256.0F.WIG C6 /r ib VSHUFPS ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX	Shuffle Packed single-precision floating-point values selected by imm8 from ymm2 and ymm3/mem.
EVEX.NDS.512.0F.W0 C6 /r ib VSHUFPS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	FV	V/V	AVX512F	Shuffle Packed single-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	Imm8
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

Moves two of the four packed single-precision floating-point values from each double qword of the first source operand (second operand) into the low quadword of each double qword of the destination operand (first operand); moves two of the four packed single-precision floating-point values from each double qword of the second source operand (third operand) into the high quadword of each double qword of the destination operand (see Figure). The selector operand (third operand) determines which values are moved to the destination operand.

EVEX encoded versions: The first source operand is a ZMM register. 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 destination operand is a ZMM register updated according to the writemask.

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 (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

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 (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

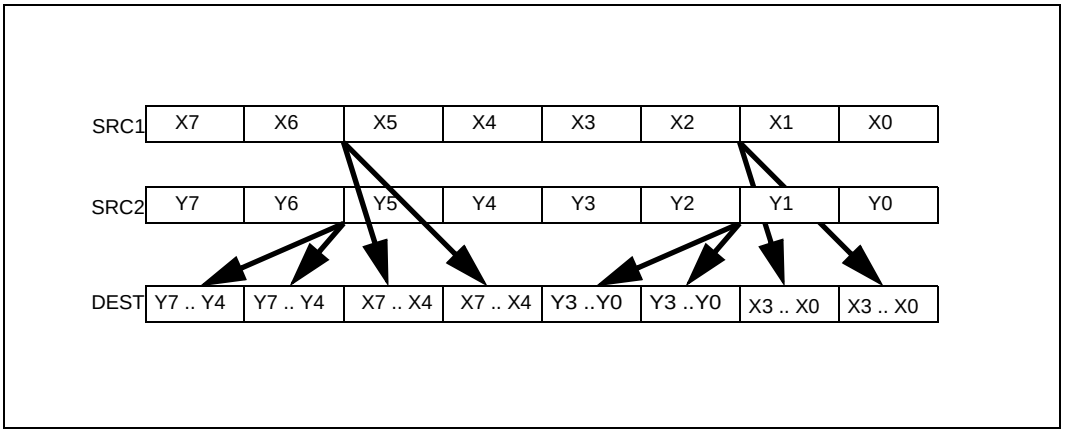


Figure 5-38. VSHUFPS Operation

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) = (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]);
    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]);
    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]);
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
```

VPSHUFPS (EVEX encoded versions when SRC2 is memory)

```
(KL, VL) = (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]);
    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]);
    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]);
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
```

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]);
```

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[MAX_VL-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[MAX_VL-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);
SHUFPS __m128 _mm_shuffle_ps (__m128 a, __m128 b, const int select);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.

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	RM	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	RM	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	RM	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.512.66.0F.W1 51/r VSQRTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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.512 encoded version: The 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 updated according to the writemask.

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

VEX.128 versions: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-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 (MAX_VL-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) = (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
```

```
            IF (EVEX.b = 1) AND (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
```

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])
```

.

VSQRTPD (VEX.128 encoded version)

```
DEST[63:0] ←SQRT(SRC[63:0])
DEST[127:64] ←SQRT(SRC[127:64])
DEST[MAX_VL-1:128] ←0
```

SQRTPD (128-bit Legacy SSE version)

```
DEST[63:0] ←SQRT(SRC[63:0])
DEST[127:64] ←SQRT(SRC[127:64])
DEST[MAX_VL-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);
SQRTPD __m128d _mm_sqrt_pd (__m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E2.

SQRTPS—Square Root of Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 51 /r SQRTPS xmm1, xmm2/m128	RM	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	RM	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	RM	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.512.0F.W0 51/r VSQRTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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 version: 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 updated according to the writemask.

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

VEX.128 versions: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-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 (MAX_VL-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) = (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
```

```
                IF (EVEX.b = 1) AND (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
```

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[MAX_VL-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[MAX_VL-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);
SQRTPS __m128 _mm_sqrt_ps (__m128 a);
```

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E2.

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/ SQRTSD xmm1,xmm2/m64	RM	V/V	SSE2	Computes square root of the low double-precision floating-point value in xmm2/m64 and stores the results in xmm1.
VEX.NDS.128.F2.0F.WIG 51/ VSQRTSD xmm1,xmm2, xmm3/m64	RVM	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.NDS.LIG.F2.0F.W1 51/ VSQRTSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	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].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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[MAX_VL-1:128] ← 0
```

VSQRTSD (VEX.128 encoded version)

```
DEST[63:0] ←SQRT(SRC2[63:0])
DEST[127:64] ←SRC1[127:64]
DEST[MAX_VL-1:128] ←0
```

SQRTSD (128-bit Legacy SSE version)

```
DEST[63:0] ←SQRT(SRC[63:0])
DEST[MAX_VL-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 Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.

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	RM	V/V	SSE	Computes square root of the low single-precision floating-point value in xmm2/m32 and stores the results in xmm1.
VEX.NDS.128.F3.0F.WIG 51 /r VSQRTSS xmm1, xmm2, xmm3/m32	RVM	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.NDS.LIG.F3.0F.W0 51 /r VSQRTSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	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].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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[127:31] ← SRC1[127:31]
DEST[MAX_VL-1:128] ← 0
```

VSQRTSS (VEX.128 encoded version)

```
DEST[31:0] ←SQRT(SRC2[31:0])
DEST[127:32] ←SRC1[127:32]
DEST[MAX_VL-1:128] ←0
```

SQRTSS (128-bit Legacy SSE version)

```
DEST[31:0] ←SQRT(SRC2[31:0])
DEST[MAX_VL-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 Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.

VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.512.66.0F3A.W0 25 /r /ib VPTERNLOGD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	FV	V/V	AVX512F	Bitwise ternary logic taking zmm1, zmm2 and zmm3/m512/m32bcst as source operands and writing the result to zmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.512.66.0F3A.W1 25 /r /ib VPTERNLOGQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	FV	V/V	AVX512F	Bitwise ternary logic taking zmm1, zmm2 and zmm3/m512/m64bcst as source operands and writing the result to zmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	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 register. The first source operand is a ZMM 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 ZMM register conditionally updated with writemask k1.

Two examples of Boolean function specified by 0xE2 and 0xE4 with the look up results of 3-bit index values are shown in Table 5-20.

Table 5-20. Examples of VPTERNLOGD/Q Imm8 Boolean Function and Input Index Values

VPTERNLOGD reg1, reg2, src3, 0xE2				VPTERNLOGD reg1, reg2, src3, 0xE4			
Bit(reg1)	Bit(reg2)	Bit(src3)	Imm8=0xE2	Bit(reg1)	Bit(reg2)	Bit(src3)	Imm8=0xE2
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

Operation

VPTERNLOGD (EVEX encoded versions)

```
(KL, VL) = (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;
```

VPTERNLOGQ (EVEX encoded versions)

```
(KL, VL) = (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;
```

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, __mmsk16 m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m512i _mm512_maskz_ternarylogic_epi32(__mmsk16 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i _mm512_ternarylogic_epi64(__m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i _mm512_mask_ternarylogic_epi64(__m512i s, __mmsk8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i _mm512_maskz_ternarylogic_epi64(__mmsk8 m, __m512i a, __m512i b, int imm);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VPTESTMD/VPTESTMQ—Logical AND and Set Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F38.W0 27 /r VPTESTMD k2 {k1}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Bitwise AND of packed doubleword integers in zmm2 and s zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FVM	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND operation on the first source operand (the second operand) and second source operand (the third operand) and stores the result in the destination operand (the first operand) under the writemask. Each bit of the test result is set to 1 if the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

The first source operand is a ZMM 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 mask register updated under the writemask.

Operation

VPTESTMD (EVEX encoded versions)

```
(KL, VL) = (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) = (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

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

See Exceptions Type E4.

VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 46 /r VPSRAVD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX2	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits.
VEX.NDS.256.66.0F38.W0 46 /r VPSRAVD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits.
EVEX.NDS.512.66.0F38.W0 46 /r VPSRAVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F38.W1 46 /r VPSRAVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in sign bits using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (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 are filled with the corresponding sign bit of the source element.

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 16 (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 (MAX_VL-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 (MAX_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The destination and first source operands are ZMM registers. The count operand can be either a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

Operation

VPSRAVD (VEX.128 version)

```
COUNT_0 ← SRC2[4 : 0]
    (* Repeat Each COUNT_i for the low 5 bits of 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[100 : 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[MAX_VL-1:128] ← 0;
```

VPSRAVD (VEX.256 version)

```
COUNT_0 ← SRC2[4 : 0];
    (* Repeat Each COUNT_i for the low 5 bits of 2nd through 8th dwords of SRC2*)
COUNT_7 ← SRC2[228 : 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[MAX_VL-1:256] ← 0;
```

VPSRAVD (EVEX encoded version)

```
(KL, VL) = (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;
```

VPSRAVQ (EVEX encoded version)

```
(KL, VL) = (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;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSRAVD __m512i _mm512_srav_epi32(__m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_mask_srav_epi32(__m512i s, __mmsk16 m, __m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_maskz_srav_epi32(__mmsk16 m, __m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_srav_epi64(__m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_mask_srav_epi64(__m512i s, __mmsk8 m, __m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_maskz_srav_epi64(__mmsk8 m, __m512i a, __m512i cnt);
VPSRAVD __m256i _mm256_srav_epi32 (__m256i m, __m256i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

PXOR/PXORD/PXORQ—Exclusive Or

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F EF /r PXOR xmm1, xmm2/m128	RM	V/V	SSE2	Bitwise XOR of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG EF /r VPXOR xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Bitwise XOR of xmm3/m128 and xmm2.
VEX.NDS.256.66.0F.WIG EF /r VPXOR ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Bitwise XOR of ymm3/m256 and ymm2.
EVEX.NDS.512.66.0F.W0 EF /r VPXORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Bitwise XOR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.NDS.512.66.0F.W1 EF /r VPXORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Bitwise XOR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical XOR operation on the second source operand and the first source operand and stores the result in the destination operand. Each bit of the result is set to 0 if the corresponding bits of the first and second operands are identical; otherwise, it is set to 0.

EVEX.512 encoded version: The first source operand is a ZMM 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 ZMM register conditionally updated with writemask k1.

VEX.256 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 (MAX_VL-1:256) of the corresponding register destination are zeroed.

VEX.128 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 (MAX_VL-1:128) of the corresponding register destination are zeroed.

Legacy SSE instructions: In 64-bit mode, 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 second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAX_VL-1:128) of the corresponding ZMM destination register remain unchanged.

Operation

PXOR (Legacy SSE instruction)

DEST[127:0] ← (DEST[127:0] BITWISE XOR SRC[127:0])

VPXOR (VEX.128 encoded instruction)

```
DEST[127:0] ← (SRC1[127:0] BITWISE XOR SRC2[127:0])
DEST[MAX_VL-1:128] ← 0
```

VPXOR (VEX.256 encoded instruction)

```
DEST[255:0] ← (SRC1[255:0] BITWISE XOR SRC2[255:0])
DEST[MAX_VL-1:256] ← 0
```

VPXORD (EVEX encoded versions)

```
(KL, VL) = (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;
```

VPXORQ (EVEX encoded versions)

```
(KL, VL) = (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;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPXORD __m512i _mm512_xor_epi32(__m512i a, __m512i b)
VPXORD __m512i _mm512_mask_xor_epi32(__m512i s, __mmsk16 m, __m512i a, __m512i b)
VPXORD __m512i _mm512_maskz_xor_epi32(__mmsk16 m, __m512i a, __m512i b)
VPXORQ __m512i _mm512_xor_epi64(__m512i a, __m512i b);
VPXORQ __m512i _mm512_mask_xor_epi64(__m512i s, __mmsk8 m, __m512i a, __m512i b);
VPXORQ __m512i _mm512_maskz_xor_epi64(__mmsk8 m, __m512i a, __m512i b);
PXOR __m128i _mm_xor_si128 (__m128i a, __m128i b)
VPXOR __m256i _mm256_xor_si256 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 4C /r VRCP14PD zmm1 {k1}{z}, zmm2/m512/m64bcst	FV	V/V	AVX512F	Computes the approximate reciprocals of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

This instruction performs a SIMD computation of the approximate reciprocals of 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 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.

Operation

VRCP14PD ((EVEX encoded versions)

```
(KL, VL) = (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;
```

Table 5-21. VRCP14PD 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

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14PD __m512d _mm512_rcp14_pd(__m512d a);
VRCP14PD __m512d _mm512_mask_rcp14_pd(__m512d s, __mmask8 k, __m512d a);
VRCP14PD __m512d _mm512_maskz_rcp14_pd(__mmask8 k, __m512d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 4D /r VRCP14SD xmm1 {k1}{z}, xmm2, xmm3/m64	T1S	V/V	AVX512F	Computes the approximate reciprocal of the scalar double-precision floating-point value in xmm3/m64 and stores the result in xmm1 using writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction performs a SIMD computation of the approximate reciprocal of the low double-precision floating-point value in the second source operand (the third operand) stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register.

The VRCP14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ 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.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Operation

VRCP14SD (EVEX version)

```
IF k1[0] OR *no writemask*
        THEN DEST[63:0] ← APPROXIMATE(1.0/SRC2[63:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

Table 5-22. 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

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14SD __m128d _mm_rcp14_sd(__m128d a, __m128d b);
VRCP14SD __m128d _mm_mask_rcp14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRCP14SD __m128d _mm_maskz_rcp14_sd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E5.

VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 4C /r VRCP14PS zmm1 {k1}{z}, zmm2/m512/m32bcst	FV	V/V	AVX512F	Computes the approximate reciprocals of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

This instruction performs a SIMD computation of the approximate reciprocals of the packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand). The maximum relative error for this approximation is less than 2^{-14}.

The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ 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.

Operation

VRCP14PS (EVEX encoded versions)

```
(KL, VL) = (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;
```

Table 5-23. VRCP14PS 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*
$X < -2^{126}$	-Underflow	Up to 18 bits of fractions are returned*
$X = 2^{-n}$	2^n	
$X = -2^{-n}$	-2^n	

* in this case the mantissa is shifted right by one or two bits

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);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 4D /r VRCP14SS xmm1 {k1}{z}, xmm2, xmm3/m32	T1S	V/V	AVX512F	Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1 using writemask k1. Also, upper double-precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction performs a SIMD computation of the approximate reciprocal of the low single-precision floating-point value in the second source operand (the third operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand 9the 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.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Operation

VRCP14SS (EVEX version)

```
IF k1[0] OR *no writemask*
        THEN DEST[31:0] ← APPROXIMATE(1.0/SRC2[31:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0
```

Table 5-24. 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*
$X < -2^{126}$	-Underflow	Up to 18 bits of fractions are returned*
$X = 2^{-n}$	2^n	
$X = -2^{-n}$	-2^n	

* in this case the mantissa is shifted right by one or two bits

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14SS __m128 _mm_rcp14_ss(__m128 a, __m128 b);
VRCP14SS __m128 _mm_mask_rcp14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRCP14SS __m128 _mm_maskz_rcp14_ss(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E5.

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.512.66.0F3A.W1 09 /r ib VRNDSCALEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	FV	V/V	AVX512F	Rounds packed double-precision floating-point values in zmm2/m512/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Round the double-precision floating-point values in the source operand by the rounding mode specified in the immediate operand 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 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

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).

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.

Immediate Control Description:

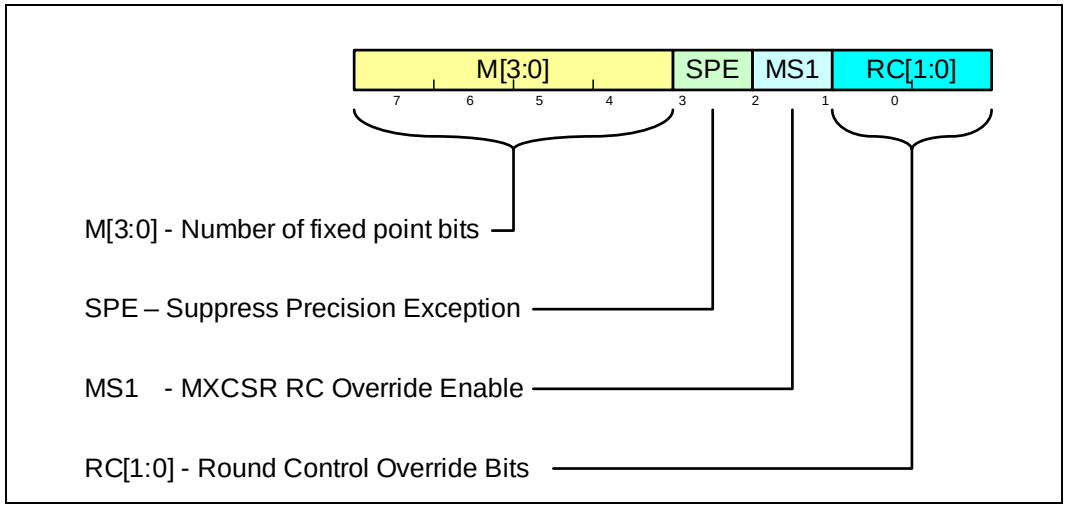


Figure 5-39. Immediate Control Description

Operation

```
RoundToIntegerPD(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) {        ; check SPE
if (SRC[63:0] != Dest[63:0]) {          ; check precision lost
set_precision()          ; set #PE
}
}
return(Dest[63:0])
}
```

VRNDSCALEPD (EVEX encoded versions)

```
(KL, VL) = (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] ← RoundToIntegerPD((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;
Zero_upper_imm[7:0] = (0h << 4 ) OR imm8[3:0];
```

Table 5-25. Immediate RC (Round Control) Bits

Bits	Field Name/Value	Description
Imm[1:0]	RC[1:0]=0	Round to nearest even
	RC[1:0]=1	Round down
	RC[1:0]=2	Round Up
	RC[1:0]=3	Truncate

Table 5-26. Immediate Control Bits

Bits	Field Name/Value	Description
Imm[2]	MS1=1	MXCSR Override Control: Use MXCSR.RC
	MS1=0	Use imm8.RC
Imm[3]	SPE=1	Suppress Precision Exceptions
	SPE=0	Use MXCSR Exception Mask
Imm[7:4]	M[3:0]	Number of fraction bits after the binary point to be preserved in the result

Table 5-27. VRNDSCALEPD Special Cases

	Returned value
Src1=±inf	Src1
Src1=±NAN	Src1 converted to QNAN
Src1=±0	Src1

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);

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Exceptions Type E2.

VRNDSCALESD—Round Scalar Float64 Value To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 0B /r ib VRNDSCALESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	T1S	V/V	AVX512F	Rounds scalar double-precision floating-point value in xmm3/m64 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

Rounds a double-precision floating-point value in the low quadword element 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 third 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 MAX_VL-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

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).

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.

Immediate Control Description:

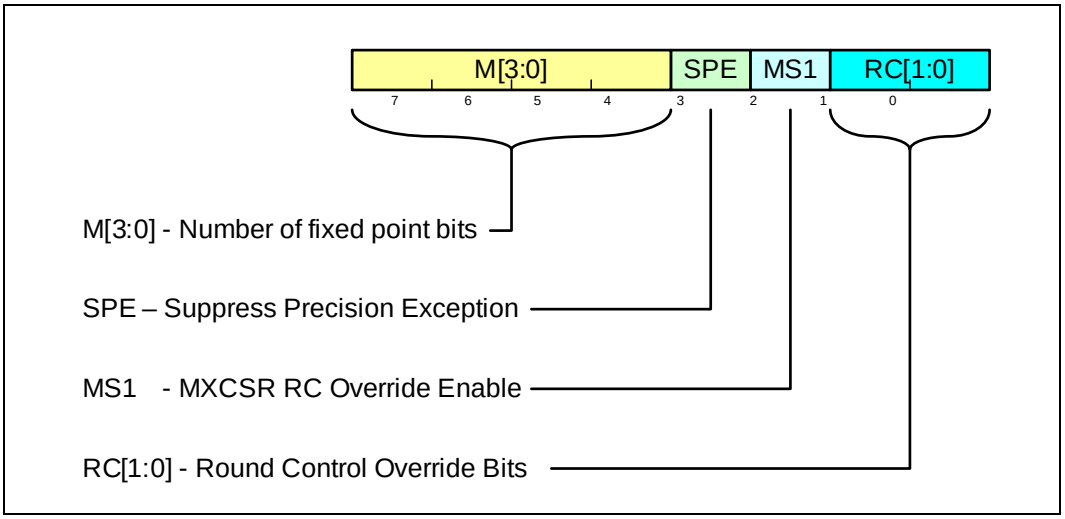


Figure 5-40. Immediate Control Description

Operation

```
RoundToIntegerPD(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) {        ; check SPE
if (SRC[63:0] != Dest[63:0]) {          ; check precision lost
set_precision()                 ; set #PE
}
}
return(Dest[63:0])
}
```

VRNDSCALESD (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN      DEST[63:0] ← RoundToIntegerPD(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[MAX_VL-1:128] ← 0
```

Table 5-28. Immediate RC (Round Control) Bits

Bits	Field Name/Value	Description
Imm[1:0]	RC[1:0]=0	Round to nearest even
	RC[1:0]=1	Round down
	RC[1:0]=2	Round Up
	RC[1:0]=3	Truncate

Table 5-29. Immediate Control Bits

Bits	Field Name/Value	Description
Imm[2]	MS1=1	MXCSR Override Control: Use MXCSR.RC
	MS1=0	Use imm8.RC
Imm[3]	SPE=1	Suppress Precision Exceptions
	SPE=0	Use MXCSR Exception Mask
Imm[7:4]	M[3:0]	Number of fraction bits after the binary point to be preserved in the result

Table 5-30. VRNDSCALESD Special Cases

	Returned value
Src1=±inf	Src1
Src1=±NAN	Src1 converted to QNAN
Src1=±0	Src1

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALESD __m128d _mm_roundscale_sd (__m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_roundscale_round_sd (__m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d _mm_mask_roundscale_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_mask_roundscale_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d _mm_maskz_roundscale_sd (__mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_maskz_roundscale_round_sd (__mmask8 k, __m128d a, __m128d b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Exceptions Type E3.

VRNDSCALEPS—Round Packed Float32 Values To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F3A.W0 08 /r ib VRNDSCALEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	FV	V/V	AVX512F	Rounds packed single-precision floating-point values in zmm2/m512/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Round the single-precision floating-point values in the source operand by the rounding mode specified in the immediate operand and places the result in the destination operand.

The destination operand (the first operand) is a ZMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPS is

ROUND(x) = 2^{-M}*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).

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.

Immediate Control Description:

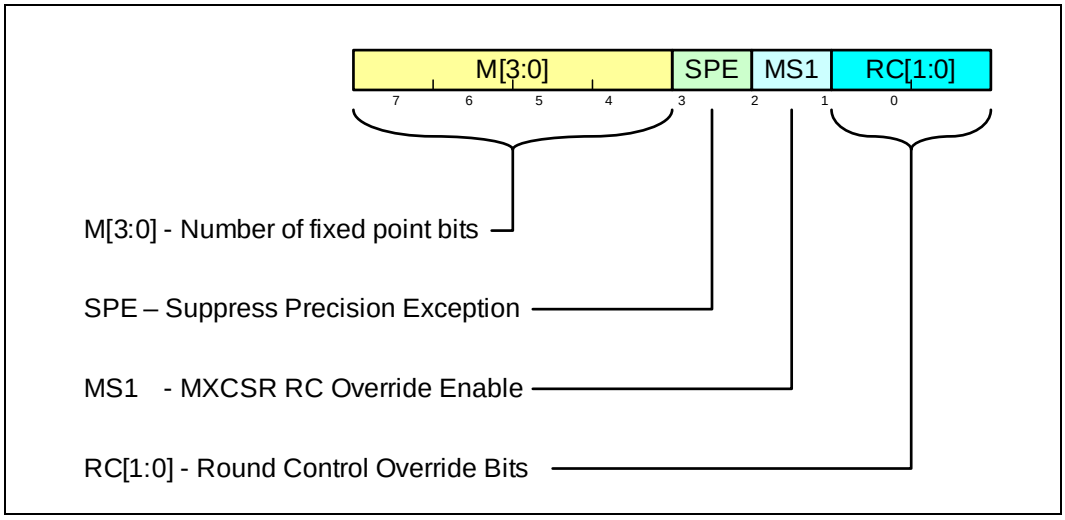


Figure 5-41. Immediate Control Description

Operation

```
RoundToIntegerPS(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) {        ; check SPE
if (SRC[31:0] != Dest[31:0]) {      ; check precision lost
set_precision()           ; set #PE
}
}
return(Dest[63:0])
}
```

VRNDSCALEPS (EVEX encoded versions)

```
(KL, VL) = (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] ← RoundToIntegerPS(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;
Zero_upper_imm[7:0] = (0h << 4 ) OR imm8[3:0];
```

Table 5-31. Immediate RC (Round Control) Bits

Bits	Field Name/Value	Description
Imm[1:0]	RC[1:0]=0	Round to nearest even
	RC[1:0]=1	Round down
	RC[1:0]=2	Round Up
	RC[1:0]=3	Truncate

Table 5-32. Immediate Control Bits

Bits	Field Name/Value	Description
Imm[2]	MS1=1	MXCSR Override Control: Use MXCSR.RC
	MS1=0	Use imm8.RC
Imm[3]	SPE=1	Suppress Precision Exceptions
	SPE=0	Use MXCSR Exception Mask
Imm[7:4]	M[3:0]	Number of fraction bits after the binary point to be preserved in the result

Table 5-33. VRNDSCALEPS Special Cases

	Returned value
Src1=±inf	Src1
Src1=±NAN	Src1 converted to QNAN
Src1=±0	Src1

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);

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Exceptions Type E2.

VRNDSCALESS—Round Scalar Float32 Value To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 0A /r ib VRNDSCALESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	T1S	V/V	AVX512F	Rounds scalar single-precision floating-point value in xmm3/m32 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Rounds the single-precision floating-point value in the low doubleword element of the second source operand (the third operand) by the rounding mode specified in the immediate operand 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 MAX_VL-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.

Immediate Control Description:

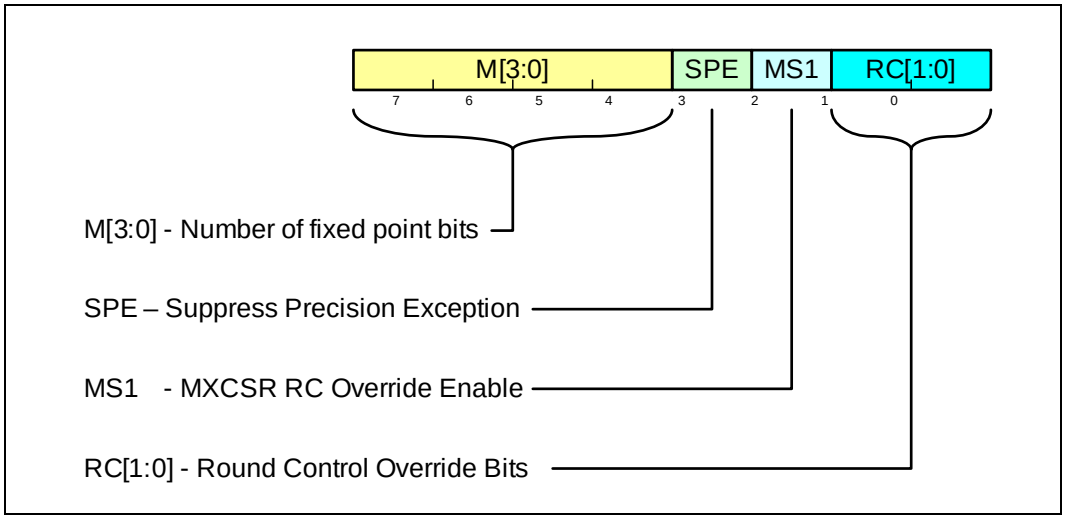


Figure 5-42. Immediate Control Description

Operation

```
RoundToIntegerPS(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) {              ; check SPE
if (SRC[31:0] != Dest[31:0]) {          ; check precision lost
    set_precision()             ; set #PE
}
}
return(Dest[63:0])
}
```

VRNDSCALESS (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundToIntegerPS(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[MAX_VL-1:128] ← 0
```

Table 5-34. Immediate RC (Round Control) Bits

Bits	Field Name/Value	Description
Imm[1:0]	RC[1:0]=0	Round to nearest even
	RC[1:0]=1	Round down
	RC[1:0]=2	Round Up
	RC[1:0]=3	Truncate

Table 5-35. Immediate Control Bits

Bits	Field Name/Value	Description
Imm[2]	MS1=1	MXCSR Override Control: Use MXCSR.RC
	MS1=0	Use imm8.RC
Imm[3]	SPE=1	Suppress Precision Exceptions
	SPE=0	Use MXCSR Exception Mask
Imm[7:4]	M[3:0]	Number of fraction bits after the binary point to be preserved in the result

Table 5-36. VRNDSCALESS Special Cases

	Returned value
Src1=±inf	Src1
Src1=±NAN	Src1 converted to QNAN
Src1=±0	Src1

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALESS __m128 _mm_roundscale_ss (__m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_roundscale_round_ss (__m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_mask_roundscale_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_mask_roundscale_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_maskz_roundscale_ss (__mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_maskz_roundscale_round_ss (__mmask8 k, __m128 a, __m128 b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Exceptions Type E3.

VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 4E /r VRSQRT14PD zmm1 {k1}{z}, zmm2/m512/m64bcst	FV	V/V	AVX512F	Computes the approximate reciprocal square roots of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1 under writemask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of the eight packed double-precision floating-point values in the source operand (the second operand) and stores the packed double-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}.

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.

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.

Operation

VRSQRT14PD (EVEX encoded versions)

```
(KL, VL) = (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;
```

Table 5-37. 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);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 4F /r VRSQRT14SD xmm1 {k1}{z}, xmm2, xmm3/m64	T1S	V/V	AVX512F	Computes the approximate reciprocal square root of the scalar double-precision floating-point value in xmm3/m64 and stores the result in the low quadword element of xmm1 using writemask k1. Bits[127:64] of xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the approximate reciprocal of the square roots of the scalar double-precision floating-point value in the low quadword element of the source operand (the second operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL-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.

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[MAX_VL-1:128] ← 0
```

Table 5-38. VRSQRT14SD Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
X = 2^{-2n}	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14SD __m128d _mm_rsqrt14_sd(__m128d a, __m128d b);
VRSQRT14SD __m128d _mm_mask_rsqrt14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRSQRT14SD __m128d _mm_maskz_rsqrt14_sd(__mmask8d m, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E5.

VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 4E /r VRSQRT14PS zmm1 {k1}{z}, zmm2/m512/m32bcst	FV	V/V	AVX512F	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of 16 packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}.

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.

The VRSQRT14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

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

Operation

VRSQRT14PS (EVEX encoded versions)

```
(KL, VL) = (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;
```

Table 5-39. 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);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 4F /r VRSQRT14SS xmm1 {k1}{z}, xmm2, xmm3/m32	T1S	V/V	AVX512F	Computes the approximate reciprocal square root of the scalar single-precision floating-point value in xmm3/m32 and stores the result in the low doubleword element of xmm1 using writemask k1. Bits[127:32] of xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

Description

Computes of the approximate reciprocal of the square root of the scalar single-precision floating-point value in the low doubleword element of the source operand (the second operand) and stores the result in the low doubleword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL-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.

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[MAX_VL-1:128] ← 0
```

Table 5-40. 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 Exceptions Type E5.

VSCALEFPD—Scale Packed Float64 Values With Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F38.W1 2C /r VSCALEFPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Scale the packed double-precision floating-point values in zmm2 using values from zmm3/m512/m64bcst. Under writemask k1.

Instruction Operand Encoding

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

Description

Performs a floating-point scale of the packed double-precision floating-point values in the first source operand by multiplying it by 2 power of the double-precision floating-point values in second source operand.

The equation of this operation is given by:

zmm1 := zmm2*$2^{floor(zmm3)}$.

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 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 with writemask k1.

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) = (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[63:0]);
```

```
                ELSE DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[i+63:i]);
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
```

Table 5-41. VSCALEFPD 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 5-42. Additional VSCALEFPD Special Cases

Special Case	Returned value	Faults
$\|result\| < 2^{-1074}$	±0 or ±Min-Denormal (Src1 sign)	Underflow
$\|result\| \geq 2^{1024}$	±INF (Src1 sign) or ±Max-normal (Src1 sign)	Overflow

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFPD __m512d _mm512_scalef_round_pd(__m512d a, __m512d b, int);
VSCALEFPD __m512d _mm512_mask_scalef_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VSCALEFPD __m512d _mm512_maskz_scalef_round_pd(__mmask8 k, __m512d a, __m512d b, int);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Exceptions Type E2.

VSCALEFSD—Scale Scalar Float64 Values With Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 2D /r VSCALEFSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Scale the scalar double-precision floating-point values in xmm2 using the value from xmm3/m64. Under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a floating-point scale of the packed double-precision floating-point value in the first source operand by multiplying it by 2 power of the double-precision floating-point value in second source operand.

The equation of this operation is given by:

$xmm1 := xmm2*2^{floor(xmm3)}$.

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.

Operation

```
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
TMP_SRC2 ← SRC2
TMP_SRC1 ← SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 64 bits floating-point value */
    DEST[63:0] ← TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
```

VSCALEFSD (EVEX encoded version)

```
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[63:0] ← SCALE(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
```

```
                DEST[63:0] ← 0
        FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

Table 5-43. VSCALEFSD 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 5-44. Additional VSCALEFSD Special Cases

Special Case	Returned value	Faults
$\|result\| < 2^{-1074}$	±0 or ±Min-Denormal (Src1 sign)	Underflow
$\|result\| \geq 2^{1024}$	±INF (Src1 sign) or ±Max-normal (Src1 sign)	Overflow

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFSD __m128d _mm_scalef_round_sd(__m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_mask_scalef_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_maskz_scalef_round_sd(__mmask8 k, __m128d a, __m128d b, int);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Exceptions Type E3.

VSCALEFPS—Scale Packed Float32 Values With Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F38.W0 2C /r VSCALEFPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Scale the packed single-precision floating-point values in zmm2 using floating-point values from zmm3/m512/m32bcst. Under writemask k1.

Instruction Operand Encoding

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

Description

Performs a floating-point scale of the packed single-precision floating-point values in the first source operand by multiplying it by 2 power of the float32 values in second source operand.

The equation of this operation is given by:

$zmm1 := zmm2*2^{floor(zmm3)}$.

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.

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.

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) = (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
```

```
                THEN DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[31:0]);
                ELSE DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[i+31:i]);
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
```

Table 5-45. VSCALEFPS 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 5-46. Additional VSCALEFPS Special Cases

Special Case	Returned value	Faults
$\|result\| < 2^{-149}$	±0 or ±Min-Denormal (Src1 sign)	Underflow
$\|result\| \geq 2^{128}$	±INF (Src1 sign) or ±Max-normal (Src1 sign)	Overflow

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFPS __m512 _mm512_scalef_round_ps(__m512 a, __m512 b, int);
VSCALEFPS __m512 _mm512_mask_scalef_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VSCALEFPS __m512 _mm512_maskz_scalef_round_ps(__mmask16 k, __m512 a, __m512 b, int);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Exceptions Type E2.

VSCALEFSS—Scale Scalar Float32 Value With Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 2D /r VSCALEFSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Scale the scalar single-precision floating-point value in xmm2 using floating-point value from xmm3/m32. Under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a floating-point scale of the scalar single-precision floating-point value in the first source operand by multiplying it by 2 power of the float32 value in second source operand.

The equation of this operation is given by:

$xmm1 := xmm2*2^{floor(xmm3)}$.

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.

Operation

```
SCALE(SRC1, SRC2)
{
            ; Check for denormal operands
TMP_SRC2 ← SRC2
TMP_SRC1 ← SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 32 bits floating-point value */
    DEST[31:0] ← TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}
```

VSCALEFSS (EVEX encoded version)

```
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← SCALE(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
```

```
                DEST[31:0] ← 0
        FI
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0
```

Table 5-47. VSCALEFSS 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 5-48. Additional VSCALEFSS Special Cases

Special Case	Returned value	Faults
$\|result\| < 2^{-149}$	±0 or ±Min-Denormal (Src1 sign)	Underflow
$\|result\| \geq 2^{128}$	±INF (Src1 sign) or ±Max-normal (Src1 sign)	Overflow

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFSS __m128 _mm_scalef_round_ss(__m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_mask_scalef_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_maskz_scalef_round_ss(__mmask8 k, __m128 a, __m128 b, int);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Exceptions Type E3.

VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 A2 /vsib VSCATTERDPS vm32z {k1}, (zmm1)	T1S	V/V	AVX512F	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A2 /vsib VSCATTERDPD vm32y {k1}, (zmm1)	T1S	V/V	AVX512F	Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W0 A3 /vsib VSCATTERQPS vm64z {k1}, (ymm1)	T1S	V/V	AVX512F	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A3 /vsib VSCATTERQPD vm64z {k1}, (zmm1)	T1S	V/V	AVX512F	Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	ModRM:reg (r)	NA	NA

Description

Stores up to 16 elements (or 8 elements) in doubleword/quadword vector zmm1 to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
- If two or more destination indices completely overlap, the "earlier" write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element after down-conversion.

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, 2 or 4 byte displacement

VSCATTERDPS (EVEX encoded versions)

```
(KL, VL)= (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)= (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)= (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)= (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);

SIMD Floating-Point Exceptions

Invalid, Overflow, Underflow, Precision, Denormal

Other Exceptions

See Exceptions Type E12.

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	RM	V/V	SSE2	Subtract packed double-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.NDS.128.66.0F.WIG 5C /r VSUBPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Subtract packed double-precision floating-point values in xmm3/mem from xmm2 and store result in xmm1.
VEX.NDS.256.66.0F.WIG 5C /r VSUBPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Subtract packed double-precision floating-point values in ymm3/mem from ymm2 and store result in ymm1.
EVEX.NDS.512.66.0F.W1 5C /r VSUBPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Subtract packed double-precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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 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 (MAX_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding register destination are unmodified.

Operation

VSUBPD (EVEX encoded versions) when src2 operand is a vector register

```
(KL, VL) = (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
```

```
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← 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
```

VSUBPD (EVEX encoded versions) when src2 operand is a memory source

(KL, VL) = (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
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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);
SUBPD __m128d _mm_sub_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

SUBPS—Subtract Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 5C /r SUBPS xmm1, xmm2/m128	RM	V/V	SSE	Subtract packed single-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.NDS.128.0F.WIG 5C /r VSUBPS xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Subtract packed single-precision floating-point values in xmm3/mem from xmm2 and stores result in xmm1.
VEX.NDS.256.0F.WIG 5C /r VSUBPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Subtract packed single-precision floating-point values in ymm3/mem from ymm2 and stores result in ymm1.
EVEX.NDS.512.0F.W0 5C /r VSUBPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Subtract packed single-precision floating-point values in zmm3/m512/m32bcst from zmm2 and stores result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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 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 (MAX_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding register destination are unmodified.

Operation

VSUBPS (EVEX encoded versions) when src2 operand is a vector register

```
(KL, VL) = (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
```

```
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← 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;
```

VSUBPS (EVEX encoded versions) when src2 operand is a memory source

```
(KL, VL) = (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;
```

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[MAX_VL-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[MAX_VL-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[MAX_VL-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);
SUBPS __m128 _mm_sub_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

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	RM	V/V	SSE2	Subtract the low double-precision floating-point value in xmm2/m64 from xmm1 and store the result in xmm1.
VEX.NDS.128.F2.0F.WIG 5C /r VSUBSD xmm1,xmm2, xmm3/m64	RVM	V/V	AVX	Subtract the low double-precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F2.0F.W1 5C /r VSUBSD xmm1 {k1}{z}, xmm2, xmm3/m64	T1S	V/V	AVX512F	Subtract the low double-precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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[MAX_VL-1:128] ← 0

VSUBSD (VEX.128 encoded version)
DEST[63:0] ←SRC1[63:0] - SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAX_VL-1:128] ←0

SUBSD (128-bit Legacy SSE version)
DEST[63:0] ←DEST[63:0] - SRC[63:0]
DEST[MAX_VL-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 Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.

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	RM	V/V	SSE	Subtract the low single-precision floating-point value in xmm2/m32 from xmm1 and store the result in xmm1.
VEX.NDS.128.F3.0F.WIG 5C /r VSUBSS xmm1,xmm2, xmm3/m32	RVM	V/V	AVX	Subtract the low single-precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F3.0F.W0 5C /r VSUBSS xmm1 {k1}{z}, xmm2, xmm3/m32	T1S	V/V	AVX512F	Subtract the low single-precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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 (MAX_VL-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 (MAX_VL-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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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[MAX_VL-1:128] ← 0

VSUBSS (VEX.128 encoded version)
DEST[31:0] ←SRC1[31:0] - SRC2[31:0]
DEST[127:32] ←SRC1[127:32]
DEST[MAX_VL-1:128] ←0

SUBSS (128-bit Legacy SSE version)
DEST[31:0] ←DEST[31:0] - SRC[31:0]
DEST[MAX_VL-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 Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.

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	RM	V/V	SSE2	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.128.66.0F.WIG 2E /r VUCOMISD xmm1, xmm2/m64	RM	V/V	AVX	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LIG.66.0F.W1 2E /r VUCOMISD xmm1, xmm2/m64{sae}	T1S	V/V	AVX512F	Compare low double-precision floating-point values in xmm1 and xmm2/m64 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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 numeric 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 Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

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
0F 2E /r UCOMISS xmm1, xmm2/m32	RM	V/V	SSE	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.128.0F.WIG 2E /r VUCOMISS xmm1, xmm2/m32	RM	V/V	AVX	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LIG.0F.W0 2E /r VUCOMISS xmm1, xmm2/m32{sae}	T1S	V/V	AVX512F	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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 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.

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 Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

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	RM	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.66.0F.WIG 15 /r VUNPCKHPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.66.0F.WIG 15 /r VUNPCKHPD ymm1,ymm2, ymm3/m256	RVM	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.NDS.512.66.0F.W1 15 /r VUNPCKHPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Unpacks and Interleaves double-precision floating-point values from high quadwords of zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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 versions:

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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX 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.

Operation

VUNPCKHPD (EVEX encoded versions when SRC2 is a register)

```
(KL, VL) = (8, 512)
    TMP_DEST[63:0] ← SRC1[127:64]
    TMP_DEST[127:64] ← SRC2[127:64]
    TMP_DEST[191:128] ← SRC1[255:192]
    TMP_DEST[255:192] ← SRC2[255:192]
    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]
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
```

VUNPCKHPD (EVEX encoded version when SRC2 is memory)

```
(KL, VL) = (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;
    TMP_DEST[63:0] ← SRC1[127:64]
    TMP_DEST[127:64] ← TMP_SRC2[127:64]
    TMP_DEST[191:128] ← SRC1[255:192]
    TMP_DEST[255:192] ← TMP_SRC2[255:192]
    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]

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
```

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]
```

VUNPCKHPD (VEX.128 encoded version)

```
DEST[63:0] ←SRC1[127:64]
DEST[127:64] ←SRC2[127:64]
```

DEST[MAX_VL-1:128] ←0

UNPCKHPD (128-bit Legacy SSE version)
DEST[63:0] ←SRC1[127:64]
DEST[127:64] ←SRC2[127:64]
DEST[MAX_VL-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)
UNPCKHPD __m128d _mm_unpackhi_pd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4NF.

UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 15 /r UNPCKHPS xmm1, xmm2/m128	RM	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.0F.WIG 15 /r VUNPCKHPS xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.0F.WIG 15 /r VUNPCKHPS ymm1,ymm2,ymm3/m256	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.NDS.512.0F.W0 15 /r VUNPCKHPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the high single-precision floating-point values from the first source operand and the second source operand.

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.

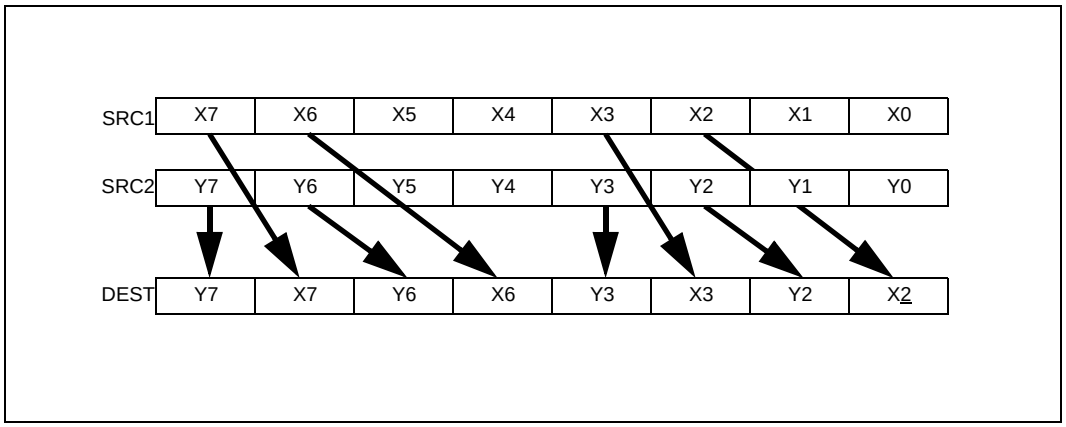


Figure 5-43. VUNPCKHPS Operation

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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

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: 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.

Operation

VUNPCKHPS (EVEX encoded version when SRC2 is a register)

```
(KL, VL) = (16, 512)
    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]
    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]
    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]

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
```

VUNPCKHPS (EVEX encoded version when SRC2 is memory)

```
(KL, VL) = (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] ← 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]
```

```
    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]
    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]
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
```

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]
```

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[MAX_VL-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[MAX_VL-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);
```

UNPCKHPS __m128 _mm_unpackhi_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4NF.

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	RM	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.66.0F.WIG 14 /r VUNPCKLPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.66.0F.WIG 14 /r VUNPCKLPD ymm1,ymm2, ymm3/m256	RVM	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.NDS.512.66.0F.W1 14 /r VUNPCKLPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Unpacks and Interleaves double-precision floating-point values from low quadwords of zmm2 and zmm3/m512/m64bcst subject to write mask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

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 versions:

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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX 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.

Operation

VUNPCKLPD (EVEX encoded versions when SRC2 is a register)

```
(KL, VL) = (8, 512)
    TMP_DEST[63:0] ← SRC1[63:0]
    TMP_DEST[127:64] ← SRC2[63:0]
    TMP_DEST[191:128] ← SRC1[191:128]
    TMP_DEST[255:192] ← SRC2[191:128]
    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]

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
```

VUNPCKLPD (EVEX encoded version when SRC2 is memory)

```
(KL, VL) = (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;
    TMP_DEST[63:0] ← SRC1[63:0]
    TMP_DEST[127:64] ← TMP_SRC2[63:0]
    TMP_DEST[191:128] ← SRC1[191:128]
    TMP_DEST[255:192] ← TMP_SRC2[191:128]
    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]
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
```

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]
```

VUNPCKLPD (VEX.128 encoded version)

```
DEST[63:0] ←SRC1[63:0]
DEST[127:64] ←SRC2[63:0]
```

DEST[MAX_VL-1:128] ←0

UNPCKLPD (128-bit Legacy SSE version)
DEST[63:0] ←SRC1[63:0]
DEST[127:64] ←SRC2[63:0]
DEST[MAX_VL-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)
UNPCKLPD __m128d _mm_unpacklo_pd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4NF.

UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 14 /r UNPCKLPS xmm1, xmm2/m128	RM	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.0F.WIG 14 /r VUNPCKLPS xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.0F.WIG 14 /r VUNPCKLPS ymm1,ymm2,ymm3/m256	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.NDS.512.0F.W0 14 /r VUNPCKLPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the low single-precision floating-point values from the first source operand and the second source operand.

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.

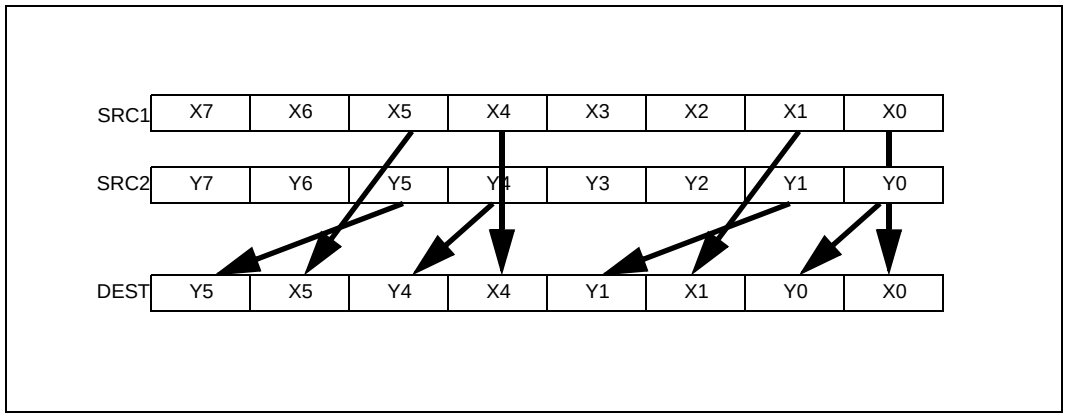


Figure 5-44. VUNPCKLPS Operation

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 (MAX_VL-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 (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX: 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.

Operation

VUNPCKLPS (EVEX encoded version when SRC2 is a ZMM register)

```
(KL, VL) = (16, 512)
    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]
    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]
    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]
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
```

VUNPCKLPS (EVEX encoded version when SRC2 is memory)

```
(KL, VL) = (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;
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]
    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]
    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]
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
```

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]
```

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[MAX_VL-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[MAX_VL-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);
UNPCKLPS __m128 _mm_unpacklo_ps (__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4NF.

CHAPTER 6
INSTRUCTION SET REFERENCE - OPMASK

Instructions for data transfer between opmask registers and between opmask/general-purpose registers are described in this chapter using the same notations and conventions listed in *Section 5.1* and *Section 5.1.5.1.*

6.1 MASK INSTRUCTIONS

KANDW —Bitwise Logical AND Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 41 /r KANDW k1, k2, k3	RVR	V/V	AVX512 F	Bitwise AND 16 bits masks k2 and k3 and place result in k1.

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 (three-operand form).

Operation

KANDW

DEST[15:0] ← SRC1[15:0] BITWISE AND SRC2[15:0]
DEST[MAX_KL-1:16] ← 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 Exceptions Type K20.

KANDNW—Bitwise Logical AND NOT Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 42 /r KANDNW k1, k2, k3	RVR	V/V	AVX512F	Bitwise AND NOT 16 bits masks k2 and k3 and place result in k1

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 (three-operand form).

Operation

KANDNW

```
DEST[15:0] ← (BITWISE NOT SRC1[15:0]) BITWISE AND SRC2[15:0]
DEST[MAX_KL-1:16] ← 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 Exceptions Type K20.

KMOVW—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	Move 16 bits mask from k2/m16 and store the result in k1.
VEX.L0.0F.W0 91 /r KMOVW m16, k1	MR	V/V	AVX512F	Move 16 bits mask from k1 and store the result in m16.
VEX.L0.0F.W0 92 /r KMOVW k1, r32	RR	V/V	AVX512F	Move 16 bits mask from r32 to k1.
VEX.L0.0F.W0 93 /r KMOVW r32, k1	RR	V/V	AVX512F	Move 16 bits mask from k1 to r32.

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 REX.W cannot be used to modify the size of the general-purpose destination.

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])
```

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 Exceptions Type K20.
Instructions with RM or MR operand encoding See Exceptions Type K21.

KUNPCKBW—Unpack for Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.66.0F.W0 4B /r KUNPCKBW k1, k2, k3	RVR	V/V	AVX512F	Unpack and interleave 8 bits masks in k2 and k3 and write word result in k1.

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 contents of second and third operands (source operands) into the low part of the first operand (destination operand). Rest of destination is zeroed.

Operation

KUNPCKBW

DEST[7:0] ← SRC2[7:0]
DEST[15:8] ← SRC1[7:0]
DEST[MAX_KL-1:16] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

KUNPCKBW __mmask16 _mm512_kunpackb(__mmask16 a, __mmask16 b);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type K20.

KNOTW—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	AVX512 F	Bitwise NOT of 16 bits mask k2.

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
```

Intel C/C++ Compiler Intrinsic Equivalent

KNOTW __mmask16 _mm512_knot(__mmask16 a);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type K20.

KORW—Bitwise Logical OR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 45 /r KORW k1, k2, k3	RVR	V/V	AVX512F	Bitwise OR 16 bits masks k2 and k3 and place result in k1.

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

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 Exceptions Type K20.

KORTESTW—OR Masks And Set Flags

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 98 /r KORTESTW k1, k2	RR	V/V	AVX512 F	Bitwise OR 16 bits masks k1 and k2 and update ZF and CF accordingly.

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;
```

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 Exceptions Type K20.

KSHIFTLW—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	Shift left 16 bits in k2 by immediate and write result in k1.

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 left the bits in the second operand (source operand) by the count specified in immediate and place result the destination operand. The rest of the destination is zeroed. 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;
```

Intel C/C++ Compiler Intrinsic Equivalent

Compiler auto generates KSHIFTLW when needed.

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type K20.

KSHIFTRW—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	AVX512 F	Shift right 16 bits in k2 by immediate and write result in k1.

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 right the bits in the second operand (source operand) by the count specified in immediate and place result the destination operand. The rest of the destination is zeroed. 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;
```

Intel C/C++ Compiler Intrinsic Equivalent

Compiler auto generates KSHIFTRW when needed.

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type K20.

KXNORW—Bitwise Logical XNOR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 46 /r KXNORW k1, k2, k3	RVR	V/V	AVX512F	Bitwise XNOR 16 bits masks k2 and k3 and place result in k1.

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

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 Exceptions Type K20.

KXORW—Bitwise Logical XOR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 47 /r KXORW k1, k2, k3	RVR	V/V	AVX512 F	Bitwise XOR 16 bits masks k2 and k3 and place result in k1.

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

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 Exceptions Type K20.

CHAPTER 7
ADDITIONAL 512-BIT INSTRUCTION EXTENSIONS

This chapter describes additional 512-bit instruction extensions to accelerate specific application domain, such as histogram operations, certain transcendental mathematic computations, or specific prefetch operations. These instructions operate on 512-bit ZMM, support opmask registers, are encoded using the same EVEX prefix encoding format, and require the same operating system support as AVX-512 Foundation instructions. The application programming model described in Chapter 2 also applies. Instructions described in this chapter follow the general documentation convention established in *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A* and *2B*. Additional notations and conventions adopted in this document are listed in *Section 5.1. Section 5.1.5.1* covers supplemental information that applies to a specific subset of instructions.

The instructions VPCONFLICT/VPLZCNT/VPTESTNM/VPBROADCASTM, also known as AVX-512 Conflict Detection instructions, can be useful for accelerating computations typically found with histogram operations. The instructions VEXP2PD/VEXP2PS/VRCP28xx/VRSQRT28xx, also known as AVX-512 Exponential and Reciprocal instructions, provide building blocks for accelerating certain transcendental math computations. The instructions VGATHERPF0xxx/VGATHERPF1xxx/VSCATTERPF0xxx/VSCATTERPF1xxx, AVX-512 Prefetch instructions, can be useful for reducing memory operation latency exposure that involve gather/scatter instructions.

7.1 DETECTION OF 512-BIT INSTRUCTION EXTENSIONS

Processor support of the AVX-512 Conflict Detection instructions are indicated by querying the feature flag:

- If CPUID.(EAX=07H, ECX=0):EBX.AVX512CD[bit 28] = 1, the family of VPCONFLICT/VPLZCNT/VPTESTNM/VPBROADCASTM instructions are supported.

Detection of these EVEX-encoded instructions operating on ZMM states and opmask registers need to follow the procedural flow in Table 7-1.

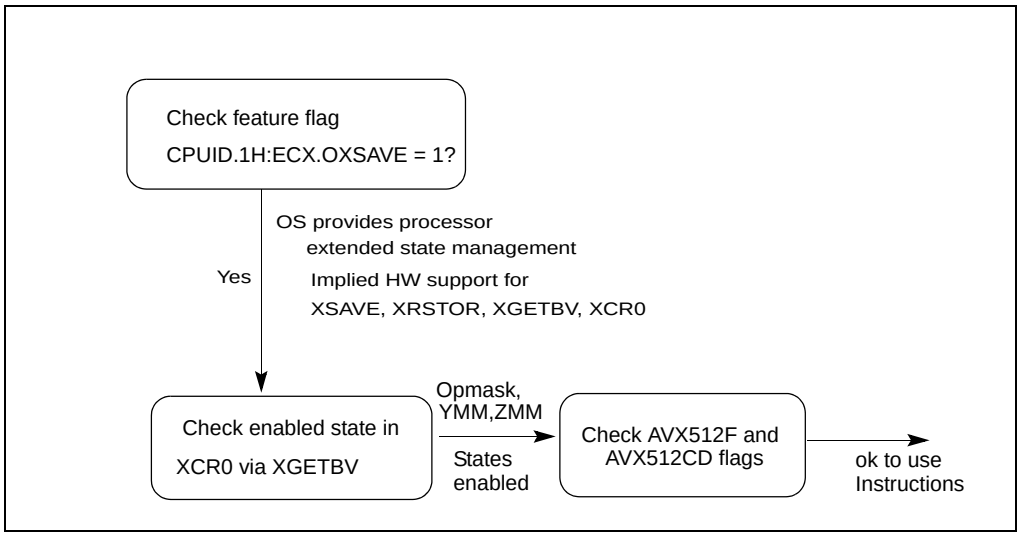


Figure 7-1. Procedural Flow of Application Detection of 512-bit Instructions

Prior to using the AVX-512 Conflict Detection instructions, the application must identify that the operating system supports the XGETBV instruction, the ZMM register state, in addition to processor's support for ZMM state management using XSAVE/XRSTOR and AVX-512 Foundation instructions. The following simplified sequence accomplishes both and is strongly recommended.

1) Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use)

2) Execute XGETBV and verify that XCR0[7:5] = '111b' (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = '11b' (XMM state and YMM state are enabled by OS).

3) Verify both CPUID.0x7.0:EBX.AVX512F[bit 16] = 1, CPUID.0x7.0:EBX.AVX512ER[bit 28] = 1.

Processor support of the AVX-512 Exponential and Reciprocal instructions are indicated by querying the feature flag:

- If CPUID.(EAX=07H, ECX=0):EBX.AVX512ER[bit 27] = 1, the collection of VEXP2PD/VEXP2PS/VRCP28xx/VRSQRT28xx instructions are supported

Processor support of the AVX-512 Prefetch instructions are indicated by querying the feature flag:

- If CPUID.(EAX=07H, ECX=0):EBX.AVX512PF[bit 26] = 1, a collection of VGATHERPF0xxx/VGATHERPF1xxx/VSCATTERPF0xxx/VSCATTERPF1xxx instructions are supported.

Detection of 512-bit instructions operating on ZMM states and opmask registers, outside of AVX-512 Foundation, need to follow the general procedural flow in Figure 7-2.

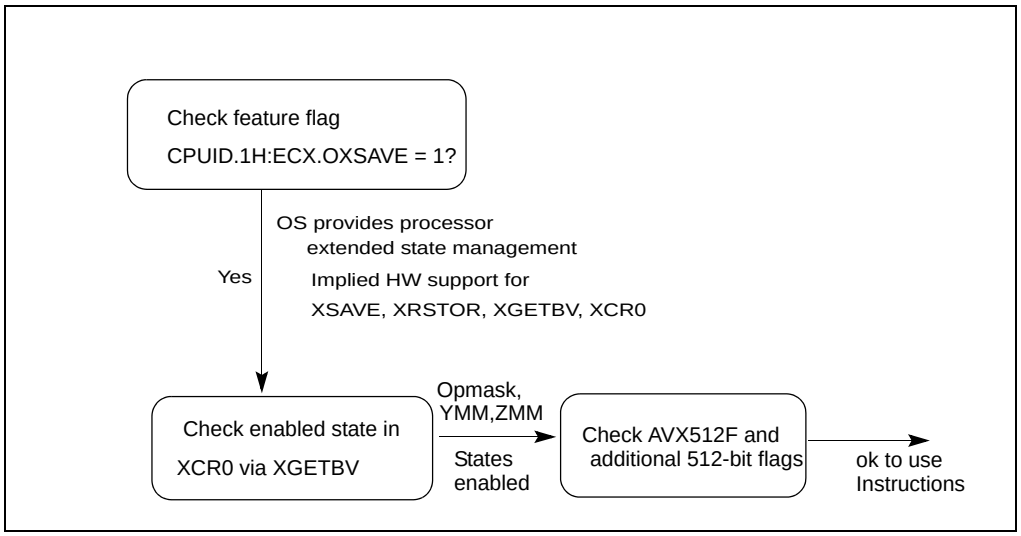


Figure 7-2. Procedural Flow of Application Detection of 512-bit Instructions

Procedural Flow of Application Detection of other 512-bit extensions:

Prior to using the AVX-512 Exponential and Reciprocal instructions, the application must identify that the operating system supports the XGETBV instruction, the ZMM register state, in addition to processor's support for ZMM state management using XSAVE/XRSTOR and AVX-512 Foundation instructions. The following simplified sequence accomplishes both and is strongly recommended.

1) Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use)

2) Execute XGETBV and verify that XCR0[7:5] = '111b' (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = '11b' (XMM state and YMM state are enabled by OS).

3) Verify both CPUID.0x7.0:EBX.AVX512F[bit 16] = 1, and CPUID.0x7.0:EBX.AVX512ER[bit 27] = 1.

Prior to using the AVX-512 Prefetch instructions, the application must identify that the operating system supports the XGETBV instruction, the ZMM register state, in addition to processor's support for ZMM state management using XSAVE/XRSTOR and AVX-512 Foundation instructions. The following simplified sequence accomplishes both and is strongly recommended.

1) Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use)

2) Execute XGETBV and verify that XCR0[7:5] = '111b' (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = '11b' (XMM state and YMM state are enabled by OS).

3) Verify both CPUID.0x7.0:EBX.AVX512F[bit 16] = 1, and CPUID.0x7.0:EBX.AVX512PF[bit 26] = 1.

7.2 INSTRUCTION SET REFERENCE

VPCONFLICTD/Q—Detect Conflicts Within a Vector of Packed Dword, Packed Qword Values into Dense Memory / Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C4 /r VPCONFLICTD zmm1 {k1}{z}, zmm2/m512/m32bcst	FV	V/V	AVX512CD	Detect duplicate double-word values in zmm2/m512/m32bcst using writemask k1.
EVEX.512.66.0F38.W1 C4 /r VPCONFLICTQ zmm1 {k1}{z}, zmm2/m512/m64bcst	FV	V/V	AVX512CD	Detect duplicate quad-word values in zmm2/m512/m64bcst using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The instruction tests each dword or 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.

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

Operation

VPCONFLICTD

```
(KL, VL) = (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
        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
```

VPCONFLICTQ

```
(KL, VL) = (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
                    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
```

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);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

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.512.66.0F38.W0 44 /r VPLZCNTD zmm1 {k1}{z}, zmm2/m512/m32bcst	FV	V/V	AVX512CD	Count the number of leading zero bits in each dword element of zmm2/m512/m32bcst using writemask k1.
EVEX.512.66.0F38.W1 44 /r VPLZCNTQ zmm1 {k1}{z}, zmm2/m512/m64bcst	FV	V/V	AVX512CD	Count the number of leading zero bits in each qword element of zmm2/m512/m64bcst using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Counts the number of leading most significant zero bits in each dword or qword element of the source operand (the second operand) and stores the results in the destination register (the first operand) according to the writemask. If an element is zero, the result for that element is the operand size.

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

Operation

VPLZCNTD

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j*32
    IF MaskBit(j) OR *no writemask*
        THEN
            temp ← 31
            DEST[i+31:i] ← 0
            WHILE (temp >= 0) AND (SRC[i+temp] = 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
```

VPLZCNTQ

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j*64
    IF MaskBit(j) OR *no writemask*
        THEN
```

```
            temp ← 63
            DEST[i+63:i] ← 0
            WHILE (temp >= 0) AND (SRC[i+temp] = 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
```

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);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

VPTESTNMD/Q—Logical AND NOT and Set

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID	Description
EVEX.NDS.512.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512CD	Bitwise AND NOT 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.NDS.512.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, zmm2, zmm3/ B_{64}(mV)	FV	V/V	AVX512CD	Bitwise AND NOT of packed quadword integers in zmm2 and zmm3/mV and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FVM	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND NOT operation on the 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 first and second operands is zero; otherwise it is set to 0Each bit of the result is set to 0 if the corresponding bits of the first and second operands are 1; otherwise, it is set to 1.

The first source operand is a ZMM registers. The second source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location.

Operation

VPTESTNMD

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j*32
    IF MaskBit(j) OR *no writemask*
        THEN
            DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] = 0)? 1 : 0
        ELSE DEST[j] ← 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VPTESTNMQ

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j*64
    IF MaskBit(j) OR *no writemask*
        THEN
            DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] = 0)? 1 : 0
        ELSE DEST[j] ← 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPTESTNMD __mmask16 _mm512_testn_epi32_mask(__m512i a, __m512i b);
VPTESTNMD __mmask16 _mm512_mask_testn_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTNMQ __mmask8 _mm512_testn_epi64_mask(__m512i a, __m512i b);
VPTESTNMQ __mmask8 _mm512_mask_testn_epi64_mask(__mmask8, __m512i a, __m512i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VPBROADCASTM—Broadcast Mask to Vector Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.F3.0F38.W1 2A /r VPBROADCASTMB2Q zmm1, k1	RM	V/V	AVX512CD	Broadcast low byte value in k1 to eight locations in zmm1.
EVEX.512.F3.0F38.W0 3A /r VPBROADCASTMW2D zmm1, k1	RM	V/V	AVX512CD	Broadcast low word value in k1 to sixteen locations in zmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Broadcasts the zero-extended 64/32 bit value of the low byte/word of the source operand (the second operand) to each 64/32 bit element of the destination operand (the first operand). The source operand is an opmask register, the destination operand is a ZMM register.

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

Operation

VPBROADCASTMB2Q

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j*64
    DEST[i+63:i] ← ZeroExtend(SRC[7:0])
ENDFOR
```

VPBROADCASTMW2D

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j*32
    DEST[i+31:i] ← ZeroExtend(SRC[15:0])
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTMB2Q __m512i _mm512_broadcastmb_epi64(__mmask8);

VPBROADCASTMW2D __m512i _mm512_broadcastmw_epi32(__mmask16);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6NF.

VEXP2PD—Approximation to the Exponential 2^x of Packed Double-Precision Floating-Point Values with Less Than 2^-23 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 C8 /r VEXP2PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	FV	V/V	AVX512ER	Computes approximations to the exponential 2^x (with less than 2^-23 of maximum relative error) of the packed double-precision floating-point values from zmm2/m512/m64bcst and stores the floating-point result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Computes the approximate base-2 exponential evaluation of the double-precision floating-point values in the source operand (the second operand) and stores the results to the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than 2^-23 of relative error.

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

Operation

VEXP2PD

```
(KL, VL) = (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] ← EXP2_23_DP(SRC[63:0])
                ELSE DEST[i+63:i] ← EXP2_23_DP(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;
```

Table 7-1. Special Values Behavior

Source Input	Result	Comments
NaN	QNaN(src)	If (SRC = SNaN) then #I
+∞	+∞	#O
+/-0	1.0f	*Exact result*
-∞	+0.0f	
Integral value N	2^ (N)	*Exact result*

Intel C/C++ Compiler Intrinsic Equivalent

VEXP2PD __m512d _mm512_exp2a23_round_pd (__m512d a, int sae);
VEXP2PD __m512d _mm512_mask_exp2a23_round_pd (__m512d a, __mmask8 m, __m512d b, int sae);
VEXP2PD __m512d _mm512_maskz_exp2a23_round_pd (__mmask8 m, __m512d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow

Other Exceptions

See Exceptions Type E2.

VEXP2PS—Approximation to the Exponential 2^x of Packed Single-Precision Floating-Point Values with Less Than 2^-23 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C8 /r VEXP2PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	FV	V/V	AVX512ER	Computes approximations to the exponential 2^x (with less than 2^-23 of maximum relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores the floating-point result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Computes the approximate base-2 exponential evaluation of the single-precision floating-point values in the source operand (the second operand) and store the results in the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than 2^-23 of relative error.

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 using writemask k1.

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

Operation

VEXP2PS

```
(KL, VL) = (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] ← EXP2_23_SP(SRC[31:0])
                ELSE DEST[i+31:i] ← EXP2_23_SP(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;
```

Table 7-2. Special Values Behavior

Source Input	Result	Comments
NaN	QNaN(src)	If (SRC = SNaN) then #I
+∞	+∞	#O
+/-0	1.0f	*Exact result*
-∞	+0.0f	
Integral value N	2^ (N)	*Exact result*

Intel C/C++ Compiler Intrinsic Equivalent

VEXP2PS __m512 _mm512_exp2a23_round_ps (__m512 a, int sae);
VEXP2PS __m512 _mm512_mask_exp2a23_round_ps (__m512 a, __mmask16 m, __m512 b, int sae);
VEXP2PS __m512 _mm512_maskz_exp2a23_round_ps (__mmask16 m, __m512 b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow

Other Exceptions

See Exceptions Type E2.

VRCP28PD—Approximation to the Reciprocal of Packed Double-Precision Floating-Point Values with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 CA /r VRCP28PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	FV	V/V	AVX512ER	Computes the approximate reciprocals (< 2^-28 relative error) of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Computes the reciprocal approximation of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, 0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

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

Operation

VRCP28PD ((EVEX encoded versions)

```
(KL, VL) = (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] ← RCP_28_DP(1.0/SRC[63:0]);
                ELSE DEST[i+63:i] ← RCP_28_DP(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;
```

Table 7-3. VRCP28PD Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \le X \le 2^{-1024}$	INF	Very small denormal
$-2^{-1024} \le X \le -0$	-INF	Very small denormal
$X > 2^{1022}$	+0.0f	
$X < -2^{1022}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^{n}	Exact result
$X = -2^{-n}$	-2^{n}	Exact result

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28PD __m512d _mm512_rcp28_round_pd (__m512d a, int sae);
VRCP28PD __m512d _mm512_mask_rcp28_round_pd(__m512d a, __mmask8 m, __m512d b, int sae);
VRCP28PD __m512d _mm512_maskz_rcp28_round_pd(__mmask8 m, __m512d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow

Other Exceptions

See Exceptions Type E2.

VRCP28SD—Approximation to the Reciprocal of Scalar Double-Precision Floating-Point Value with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 CB /r VRCP28SD xmm1 {k1}{z}, xmm2, xmm3/m64 {sae}	T1S	V/V	AVX512ER	Computes the approximate reciprocal (< 2^-28 relative error) of the scalar double-precision floating-point value in xmm3/m64 and stores the results in xmm1. Under writemask. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Computes the reciprocal approximation of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error. The result is written into the low float64 element of the destination operand according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, 0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VRCP28SD ((EVEX encoded versions)

```
IF k1[0] OR *no writemask* THEN
        DEST[63: 0] ← RCP_28_DP(1.0/SRC1[63: 0]);
ELSE
    IF *merging-masking*            ; merging-masking
        THEN *DEST[63: 0] remains unchanged*
        ELSE                        ; zeroing-masking
            DEST[63: 0] ← 0
    FI;
FI;
ENDFOR;
DEST[127:64] ← SRC2[127: 64]
DEST[MAX_VL-1:128] ← 0
```

Table 7-4. VRCP28SD Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \le X \le 2^{-1024}$	INF	Very small denormal
$-2^{-1024} \le X \le -0$	-INF	Very small denormal
$X > 2^{1022}$	+0.0f	
$X < -2^{1022}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^{n}	Exact result
$X = -2^{-n}$	-2^{n}	Exact result

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28SD __m128d _mm_rcp28_round_sd (__m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_mask_rcp28_round_sd(__m128d s, __mmask8 m, __m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_maskz_rcp28_round_sd(__mmask8 m, __m128d a, __m128d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow

Other Exceptions

See Exceptions Type E3.

VRCP28PS—Approximation to the Reciprocal of Packed Single-Precision Floating-Point Values with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 CA /r VRCP28PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	FV	V/V	AVX512ER	Computes the approximate reciprocals (< 2^-28 relative error) of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Computes the reciprocal approximation of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand) using the writemask k1. The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final results are rounded to < 2^-23 relative error before written to the destination.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, 0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

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 using writemask k1.

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

Operation

VRCP28PD ((EVEX encoded versions)

```
(KL, VL) = (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] ← RCP_28_SP(1.0/SRC[31:0]);
                ELSE DEST[i+31:i] ← RCP_28_SP(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;
```

Table 7-5. VRCP28PS Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \le X \le 2^{-128}$	INF	Very small denormal
$-2^{-128} \le X \le -0$	-INF	Very small denormal
$X > 2^{126}$	+0.0f	
$X < -2^{126}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^{n}	Exact result
$X = -2^{-n}$	-2^{n}	Exact result

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28PS _mm512_rcp28_round_ps (__m512 a, int sae);
VRCP28PS __m512 _mm512_mask_rcp28_round_ps(__m512 s, __mmask16 m, __m512 a, int sae);
VRCP28PS __m512 _mm512_maskz_rcp28_round_ps(__mmask16 m, __m512 a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow

Other Exceptions

See Exceptions Type E2.

VRCP28SS—Approximation to the Reciprocal of Scalar Single-Precision Floating-Point Value with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 CB /r VRCP28SS xmm1 {k1}{z}, xmm2, xmm3/m32 {sae}	T1S	V/V	AVX512ER	Computes the approximate reciprocal (< 2^-28 relative error) of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Under writemask. Also, upper 3 single-precision floating-point values (bits[127:32]) from xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Computes the reciprocal approximation of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final result is rounded to < 2^-23 relative error before written into the low float32 element of the destination according to writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, 0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VRCP28SS ((EVEX encoded versions)

```
IF k1[0] OR *no writemask* THEN
        DEST[31: 0] ← RCP_28_SP(1.0/SRC1[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] ← SRC2[127: 32]
DEST[MAX_VL-1:128] ← 0
```

Table 7-6. VRCP28SD Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \le X \le 2^{-1024}$	INF	Very small denormal
$-2^{-1024} \le X \le -0$	-INF	Very small denormal
$X > 2^{1022}$	+0.0f	
$X < -2^{1022}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^{n}	Exact result
$X = -2^{-n}$	-2^{n}	Exact result

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28SS __m128 _mm_rcp28_round_ss (__m128 a, __m128 b, int sae);
VRCP28SS __m128 _mm_mask_rcp28_round_ss(__m128 s, __mmask8 m, __m128 a, __m128 b, int sae);
VRCP28SS __m128 _mm_maskz_rcp28_round_ss(__mmask8 m, __m128 a, __m128 b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow

Other Exceptions

See Exceptions Type E3.

VRSQRT28PD—Approximation to the Reciprocal Square Root of Packed Double-Precision Floating-Point Values with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 CC /r VRSQRT28PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	FV	V/V	AVX512ER	Computes approximations to the Reciprocal square root (<2^-28 relative error) of the packed double-precision floating-point values from zmm2/m512/m64bcst and stores result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Computes the reciprocal square root of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

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

Operation

VRSQRT28PD (EVEX encoded versions)

```
(KL, VL) = (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] ← (1.0/ SQRT(SRC[63:0]));
                ELSE DEST[i+63:i] ← (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;
```

Table 7-7. VRSQRT28PD Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
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

VRSQRT28PD __m512d _mm512_rsqrt28_round_pd(__m512d a, int sae);
VRSQRT28PD __m512d _mm512_mask_rsqrt28_round_pd(__m512d s, __mmask8 m,__m512d a, int sae);
VRSQRT28PD __m512d _mm512_maskz_rsqrt28_round_pd(__mmask8 m,__m512d a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow

Other Exceptions

See Exceptions Type E2.

VRSQRT28SD—Approximation to the Reciprocal Square Root of Scalar Double-Precision Floating-Point Value with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 CD /r VRSQRT28SD xmm1 {k1}{z}, xmm2, xmm3/m64 {sae}	T1S	V/V	AVX512ER	Computes approximate reciprocal square root (<2^-28 relative error) of the scalar double-precision floating-point value from xmm3/m64 and stores result in xmm1with writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the reciprocal square root of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than 2^-28 of maximum relative error. The result is written into the low float64 element of xmm1 according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register.

Operation

VRSQRT28SD (EVEX encoded versions)

```
    IF k1[0] OR *no writemask* THEN
                DEST[63: 0] ← (1.0/ SQRT(SRC[63: 0]));
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63: 0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63: 0] ← 0
        FI;
    FI;
ENDFOR;
DEST[127:64] ← SRC2[127: 64]
DEST[MAX_VL-1:128] ← 0
```

Table 7-8. VRSQRT28SD Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
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

VRSQRT28SD __m128d _mm_rsqrt28_round_sd(__m128d a, __m128b b, int sae);
VRSQRT28SD __m128d _mm_mask_rsqrt28_round_pd(__m128d s, __mmask8 m,__m128d a, __m128d b, int sae);
VRSQRT28SD __m128d _mm_maskz_rsqrt28_round_pd(__mmask8 m,__m128d a, __m128d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow

Other Exceptions

See Exceptions Type E3.

VRSQRT28PS—Approximation to the Reciprocal Square Root of Packed Single-Precision Floating-Point Values with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 CC /r VRSQRT28PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	FV	V/V	AVX512ER	Computes approximations to the Reciprocal square root (<2^-28 relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Computes the reciprocal square root of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final results is rounded to < 2^-23 relative error before written to the destination.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

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 using writemask k1.

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

Operation

VRSQRT28PS (EVEX encoded versions)

```
(KL, VL) = (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] ← (1.0/ SQRT(SRC[31:0]));
                ELSE DEST[i+31:i] ← (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;
```

Table 7-9. VRSQRT28PS Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
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

VRSQRT28PS __m512 _mm512_rsqrt28_round_ps(__m512 a, int sae);
VRSQRT28PS __m512 _mm512_mask_rsqrt28_round_ps(__m512 s, __mmask16 m,__m512 a, int sae);
VRSQRT28PS __m512 _mm512_maskz_rsqrt28_round_ps(__mmask16 m,__m512 a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow

Other Exceptions

See Exceptions Type E2.

VRSQRT28SS—Approximation to the Reciprocal Square Root of Scalar Single-Precision Floating-Point Value with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 CD /r VRSQRT28SS xmm1 {k1}{z}, xmm2, xmm3/m32 {sae}	T1S	V/V	AVX512ER	Computes approximate reciprocal square root (<2^-28 relative error) of the scalar single-precision floating-point value from xmm3/m32 and stores result in xmm1with writemask k1. Also, upper 3 single-precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the reciprocal square root of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final result is rounded to < 2^-23 relative error before written to the low float32 element of the destination according to the writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register.

Operation

VRSQRT28SS (EVEX encoded versions)

```
    IF k1[0] OR *no writemask* THEN
                DEST[31: 0] ← (1.0/ SQRT(SRC[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] ← SRC2[127: 32]
DEST[MAX_VL-1:128] ← 0
```

Table 7-10. VRSQRT28SS Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
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

VRSQRT28SS __m128 _mm_rsqrt28_round_ss(__m128 a, __m128 b, int sae);
VRSQRT28SS __m128 _mm512_mask_rsqrt28_round_ss(__m128 s, __mmask8 m,__m128 a,__m128 b, int sae);
VRSQRT28SS __m128 _mm512_maskz_rsqrt28_round_ss(__mmask8 m,__m128 a,__m128 b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow

Other Exceptions

See Exceptions Type E3.

VGATHERPF0DPS/VGATHERPF0QPS/VGATHERPF0DPD/VGATHERPF0QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /1 /vsib VGATHERPF0DPS vm32z {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W0 C7 /1 /vsib VGATHERPF0QPS vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W1 C6 /1 /vsib VGATHERPF0DPD vm32y {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W1 C7 /1 /vsib VGATHERPF0QPD vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T0 hint.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into to a location in the cache hierarchy specified by a locality hint (T0):

• T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

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, 2 or 4 byte displacement
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VGATHERPF0DPS (EVEX encoded version)

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF0DPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF0QPS (EVEX encoded version)

```
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF0QPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERPF0DPD void _mm512_mask_prefetch_i32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF0DPS void _mm512_mask_prefetch_i32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF0QPD void _mm512_mask_prefetch_i64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF0QPS void _mm512_mask_prefetch_i64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12NP.

VGATHERPF1DPS/VGATHERPF1QPS/VGATHERPF1DPD/VGATHERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /2 /vsib VGATHERPF1DPS vm32z {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W0 C7 /2 /vsib VGATHERPF1QPS vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W1 C6 /2 /vsib VGATHERPF1DPD vm32y {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W1 C7 /2 /vsib VGATHERPF1QPD vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T1 hint.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into to a location in the cache hierarchy specified by a locality hint (T1):

• T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

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, 2 or 4 byte displacement
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VGATHERPF1DPS (EVEX encoded version)

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF1DPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF1QPS (EVEX encoded version)

```
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF1QPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERPF1DPD void _mm512_mask_prefetch_i32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1DPS void _mm512_mask_prefetch_i32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF1QPD void _mm512_mask_prefetch_i64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1QPS void _mm512_mask_prefetch_i64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12NP.

VSCATTERPF0DPS/VSCATTERPF0QPS/VSCATTERPF0DPD/VSCATTERPF0QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint with Intent to Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /5 /vsib VSCATTERPF0DPS vm32z {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W0 C7 /5 /vsib VSCATTERPF0QPS vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W1 C6 /5 /vsib VSCATTERPF0DPD vm32y {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W1 C7 /5 /vsib VSCATTERPF0QPD vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T0 hint with intent to write.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

cache lines will be brought into exclusive state (RFO) specified by a locality hint (T0):

• T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

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, 2 or 4 byte displacement
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VSCATTERPF0DPS (EVEX encoded version)

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF0DPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF0QPS (EVEX encoded version)

```
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF0QPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCATTERPF0DPD void _mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPF0DPD void _mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPF0DPS void _mm512_prefetch_i32scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF0DPS void _mm512_mask_prefetch_i32scatter_ps(void *base, __mmask16 m, __m512i vdx, int scale, int hint);
VSCATTERPF0QPD void _mm512_prefetch_i64scatter_pd(void * base, __m512i vdx, int scale, int hint);
VSCATTERPF0QPD void _mm512_mask_prefetch_i64scatter_pd(void * base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF0QPS void _mm512_prefetch_i64scatter_ps(void * base, __m512i vdx, int scale, int hint);
VSCATTERPF0QPS void _mm512_mask_prefetch_i64scatter_ps(void * base, __mmask8 m, __m512i vdx, int scale, int hint);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12NP.

VSCATTERPF1DPS/VSCATTERPF1QPS/VSCATTERPF1DPD/VSCATTERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint with Intent to Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /6 /vsib VSCATTERPF1DPS vm32z {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W0 C7 /6 /vsib VSCATTERPF1QPS vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W1 C6 /6 /vsib VSCATTERPF1DPD vm32y {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W1 C7 /6 /vsib VSCATTERPF1QPD vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T1 hint with intent to write.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

cache lines will be brought into exclusive state (RFO) specified by a locality hint (T1):

• T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

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, 2 or 4 byte displacement
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VSCATTERPF1DPS (EVEX encoded version)

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF1DPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF1QPS (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF1QPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCATTERPF1DPD void _mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPF1DPD void _mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPF1DPS void _mm512_prefetch_i32scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF1DPS void _mm512_mask_prefetch_i32scatter_ps(void *base, __mmask16 m, __m512i vdx, int scale, int hint);
VSCATTERPF1QPD void _mm512_prefetch_i64scatter_pd(void * base, __m512i vdx, int scale, int hint);
VSCATTERPF1QPD void _mm512_mask_prefetch_i64scatter_pd(void * base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF1QPS void _mm512_prefetch_i64scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF1QPS void _mm512_mask_prefetch_i64scatter_ps(void *base, __mmask8 m, __m512i vdx, int scale, int hint);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12NP.

PREFETCHWT1—Prefetch Vector Data Into Caches with Intent to Write and T1 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 0D /2 PREFETCHWT1 m8	M	V/V	TBD	Move data from m8 closer to the processor using T1 hint with intent to write.

Instruction Operand Encoding

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

Description

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by an intent to write hint (so that data is brought into 'Exclusive' state via a request for ownership) and a locality hint:

• T1 (temporal data with respect to first level cache)—prefetch data into the second level cache.

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte. Use of any ModR/M value other than the specified ones will lead to unpredictable behavior.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCHh instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A PREFETCHh instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHh instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHh instruction is also unordered with respect to CLFLUSH instructions, other PREFETCHh instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

```
Prefetch (m8, Level = 1, EXCLUSIVE=1);
```

Flags Affected

All flags are affected

C/C++ Compiler Intrinsic Equivalent

void _mm_prefetch(char const *, int hint= _MM_HINT_ET1);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.

CHAPTER 8
INTEL® SHA EXTENSIONS

8.1 OVERVIEW

This chapter describes a family of instruction extensions that target the acceleration of the Secure Hash Algorithm (SHA), specifically the SHA-1 and SHA-256 variants. The instruction syntax generally has two operands (in one case there is an implicit xmm0 register operand, in another a third immediate operand), where the first operand is an XMM register that provides the source as input and is the destination storing the result as well. The second source can be an XMM register or a 16-Byte aligned 128-bit memory location. In 64-bit mode, using a REX prefix in the form REX.R permits the instructions to access additional registers (XMM8-XMM15). The SHA extensions do not update any arithmetic flags and are valid in 32 and 64-bit modes. Exception behavior of the SHA extensions follows type 4 defined in Chapter 2 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

8.2 DETECTION OF INTEL SHA EXTENSIONS

A processor supports Intel SHA extensions if CPUID.(EAX=07H, ECX=0):EBX.SHA [bit 29] = 1. The SHA extensions require only XMM state support on operating systems, similar to SSE2 instructions.

8.2.1 Common Transformations and Primitive Functions

The following primitive functions and transformations are used in the algorithmic descriptions of SHA1 and SHA256 instruction extensions SHA1NEXTE, SHA1RNDS4, SHA1MSG1, SHA1MSG2, SHA256RNDS4, SHA256MSG1 and SHA256MSG2. The operands of these primitives and transformation are generally 32-bit DWORD integers.

- f0(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 1 to 20 processing.

  f0(B,C,D) ← (B AND C) XOR ((NOT(B) AND D)

- f1(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 21 to 40 processing.

  f1(B,C,D) ← B XOR C XOR D

- f2(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 41 to 60 processing.

  f2(B,C,D) ← (B AND C) XOR (B AND D) XOR (C AND D)

- f3(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 61 to 80 processing. It is the same as f1().

  f3(B,C,D) ← B XOR C XOR D

- Ch(): A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).

  Ch(E,F,G) ← (E AND F) XOR ((NOT E) AND G)

- Maj(): A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).

  Maj(A,B,C) ← (A AND B) XOR (A AND C) XOR (B AND C)

ROR is rotate right operation

(A ROR N) ← A[N-1:0] || A[Width-1:N]

ROL is rotate left operation
(A ROL N) ← A ROR (Width-N)

SHR is the right shift operation
(A SHR N) ← ZEROES[N-1:0] || A[Width-1:N]

- Σ_0( ): A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.
  Σ_0(A) ← (A ROR 2) XOR (A ROR 13) XOR (A ROR 22)

- Σ_1( ): A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.
  Σ_1(E) ← (E ROR 6) XOR (E ROR 11) XOR (E ROR 25)

- σ_0( ): A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.
  σ_0(W) ← (W ROR 7) XOR (W ROR 18) XOR (W SHR 3)

- σ_1( ): A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.
  σ_1(W) ← (W ROR 17) XOR (W ROR 19) XOR (W SHR 10)

- K_i: SHA1 Constants dependent on immediate i.
  K0 = 0x5A827999
  K1 = 0x6ED9EBA1
  K2 = 0X8F1BBCDC
  K3 = 0xCA62C1D6

8.3 SHA EXTENSIONS REFERENCE

SHA1RNDS4—Perform Four Rounds of SHA1 Operation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 3A CC /r ib SHA1RNDS4 xmm1, xmm2/m128, imm8	RMI	V/V	SHA	Performs four rounds of SHA1 operation operating on SHA1 state (A,B,C,D) from xmm1, with a pre-computed sum of the next 4 round message dwords and state variable E from xmm2/m128. The immediate byte controls logic functions and round constants

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8

Description

The SHA1RNDS4 instruction performs four rounds of SHA1 operation using an initial SHA1 state (A,B,C,D) from the first operand (which is a source operand and the destination operand) and some pre-computed sum of the next 4 round message dwords, and state variable E from the second operand (a source operand). The updated SHA1 state (A,B,C,D) after four rounds of processing is stored in the destination operand.

Operation

SHA1RNDS4

The function f() and Constant K are dependent on the value of the immediate.

```
IF ( imm8[1:0] = 0 )
    THEN f() ← f0(), K ← K0;
ELSE IF ( imm8[1:0] = 1 )
    THEN f() ← f1(), K ← K1;
ELSE IF ( imm8[1:0] = 2 )
    THEN f() ← f2(), K ← K2;
ELSE IF ( imm8[1:0] = 3 )
    THEN f() ← f3(), K ← K3;
FI;

A ← SRC1[127:96];
B ← SRC1[95:64];
C ← SRC1[63:32];
D ← SRC1[31:0];
W0E ← SRC2[127:96];
W1 ← SRC2[95:64];
W2 ← SRC2[63:32];
W3 ← SRC2[31:0];

Round i = 0 operation:
A_1 ← f (B, C, D) + (A ROL 5) +W0E +K;
B_1 ← A;
C_1 ← B ROL 30;
D_1 ← C;
E_1 ← D;

FOR i = 1 to 3
    A_(i +1) ← f (B_i, C_i, D_i) + (A_i ROL 5) +Wi+ E_i +K;
    B_(i +1) ← A_i;
```

```
    C_(i +1) ← B_i ROL 30;
    D_(i +1) ← C_i;
    E_(i +1) ← D_i;
ENDFOR

DEST[127:96] ← A_4;
DEST[95:64] ← B_4;
DEST[63:32] ← C_4;
DEST[31:0] ← D_4;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1RNDS4: __m128i _mm_sha1rnds4_epu32(__m128i, __m128i, const int);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA1NEXTE—Calculate SHA1 State Variable E after Four Rounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 C8 /r SHA1NEXTE xmm1, xmm2/m128	RM	V/V	SHA	Calculates SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in xmm1. The calculated value of the SHA1 state variable E is added to the scheduled dwords in xmm2/m128, and stored with some of the scheduled dwords in xmm1

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA

Description

The SHA1NEXTE calculates the SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in the destination operand. The calculated value of the SHA1 state variable E is added to the source operand, which contains the scheduled dwords.

Operation

SHA1NEXTE

TMP ← (SRC1[127:96] ROL 30);

DEST[127:96] ← SRC2[127:96] + TMP;
DEST[95:64] ← SRC2[95:64];
DEST[63:32] ← SRC2[63:32];
DEST[31:0] ← SRC2[31:0];

Intel C/C++ Compiler Intrinsic Equivalent

SHA1NEXTE: __m128i _mm_sha1nexte_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA1MSG1—Perform an Intermediate Calculation for the Next Four SHA1 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 C9 /r SHA1MSG1 xmm1, xmm2/m128	RM	V/V	SHA	Performs an intermediate calculation for the next four SHA1 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA

Description

The SHA1MSG1 instruction is one of two SHA1 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA1 message dwords.

Operation

SHA1MSG1

```
W0 ← SRC1[127:96] ;
W1 ← SRC1[95:64] ;
W2 ← SRC1[63: 32] ;
W3 ← SRC1[31: 0] ;
W4 ← SRC2[127:96] ;
W5 ← SRC2[95:64] ;

DEST[127:96] ← W2 XOR W0;
DEST[95:64] ← W3 XOR W1;
DEST[63:32] ← W4 XOR W2;
DEST[31:0] ← W5 XOR W3;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1MSG1: __m128i _mm_sha1msg1_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA1MSG2—Perform a Final Calculation for the Next Four SHA1 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 CA /r SHA1MSG2 xmm1, xmm2/m128	RM	V/V	SHA	Performs the final calculation for the next four SHA1 message dwords using intermediate results from xmm1 and the previous message dwords from xmm2/m128, storing the result in xmm1

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA

Description

The SHA1MSG2 instruction is one of two SHA1 message scheduling instructions. The instruction performs the final calculation to derive the next four SHA1 message dwords.

Operation

SHA1MSG2

```
W13 ← SRC2[95:64] ;
W14 ← SRC2[63: 32] ;
W15 ← SRC2[31: 0] ;
W16 ← (SRC1[127:96] XOR W13 ) ROL 1;
W17 ← (SRC1[95:64] XOR W14) ROL 1;
W18 ← (SRC1[63: 32] XOR W15) ROL 1;
W19 ← (SRC1[31: 0] XOR W16) ROL 1;

DEST[127:96] ← W16;
DEST[95:64] ← W17;
DEST[63:32] ← W18;
DEST[31:0] ← W19;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1MSG2: __m128i _mm_sha1msg2_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA256RNDS2—Perform Two Rounds of SHA256 Operation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 CB /r SHA256RNDS2 xmm1, xmm2/m128, <XMM0>	RM0	V/V	SHA	Perform 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from xmm1, an initial SHA256 state (A,B,E,F) from xmm2/m128, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand XMM0, storing the updated SHA256 state (A,B,E,F) result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Implicit XMM0 (r)

Description

The SHA256RNDS2 instruction performs 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from the first operand, an initial SHA256 state (A,B,E,F) from the second operand, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand xmm0. Note that only the two lower dwords of XMM0 are used by the instruction.

The updated SHA256 state (A,B,E,F) is written to the first operand, and the second operand can be used as the updated state (C,D,G,H) in later rounds.

Operation

SHA256RNDS2

A_0 ← SRC2[127:96];
B_0 ← SRC2[95:64];
C_0 ← SRC1[127:96];
D_0 ← SRC1[95:64];
E_0 ← SRC2[63:32];
F_0 ← SRC2[31:0];
G_0 ← SRC1[63:32];
H_0 ← SRC1[31:0];
WK_0 ← XMM0[31: 0];
WK_1 ← XMM0[63: 32];

FOR i = 0 to 1
    A_(i +1) ← Ch (E_i, F_i, G_i) +Σ_1(E_i) +WK_i+ H_i + Maj(A_i , B_i, C_i) +Σ_0(A_i);
    B_(i +1) ← A_i;
    C_(i +1) ← B_i ;
    D_(i +1) ← C_i;
    E_(i +1) ← Ch (E_i, F_i, G_i) +Σ_1(E_i) +WK_i+ H_i + D_i;
    F_(i +1) ← E_i ;
    G_(i +1) ← F_i;
    H_(i +1) ← G_i;
ENDFOR

DEST[127:96] ← A_2;
DEST[95:64] ← B_2;
DEST[63:32] ← E_2;
DEST[31:0] ← F_2;

Intel C/C++ Compiler Intrinsic Equivalent

SHA256RNDS2: __m128i _mm_sha256rnds2_epu32(__m128i, __m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA256MSG1—Perform an Intermediate Calculation for the Next Four SHA256 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 CC /r SHA256MSG1 xmm1, xmm2/m128	RM	V/V	SHA	Performs an intermediate calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA

Description

The SHA256MSG1 instruction is one of two SHA256 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA256 message dwords.

Operation

SHA256MSG1

W4 ← SRC2[31: 0] ;
W3 ← SRC1[127:96] ;
W2 ← SRC1[95:64] ;
W1 ← SRC1[63: 32] ;
W0 ← SRC1[31: 0] ;

DEST[127:96] ← W3 + σ_0(W4);
DEST[95:64] ← W2 + σ_0(W3);
DEST[63:32] ← W1 + σ_0(W2);
DEST[31:0] ← W0 + σ_0(W1);

Intel C/C++ Compiler Intrinsic Equivalent

SHA256MSG1: __m128i _mm_sha256msg1_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA256MSG2—Perform a Final Calculation for the Next Four SHA256 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 CD /r SHA256MSG2 xmm1, xmm2/m128	RM	V/V	SHA	Performs the final calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA

Description

The SHA256MSG2 instruction is one of two SHA2 message scheduling instructions. The instruction performs the final calculation for the next four SHA256 message dwords.

Operation

SHA256MSG2

W14 ← SRC2[95:64] ;
W15 ← SRC2[127:96] ;
W16 ← SRC1[31: 0] + σ_1(W14) ;
W17 ← SRC1[63: 32] + σ_1(W15) ;
W18 ← SRC1[95: 64] + σ_1(W16) ;
W19 ← SRC1[127: 96] + σ_1(W17) ;

DEST[127:96] ← W19 ;
DEST[95:64] ← W18 ;
DEST[63:32] ← W17 ;
DEST[31:0] ← W16;

Intel C/C++ Compiler Intrinsic Equivalent

SHA256MSG2 : __m128i _mm_sha256msg2_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

This page was intentionally left blank.

CHAPTER 9
INTEL® MEMORY PROTECTION EXTENSIONS

9.1 INTEL® MEMORY PROTECTION EXTENSIONS (INTEL® MPX)

Intel® Memory Protection Extensions (Intel® MPX) is a new capability introduced into Intel Architecture. Intel MPX can increase the robustness of software when it is used in conjunction with compiler changes to check memory references, for those references whose compile-time normal intentions are usurped at runtime due to buffer overflow or underflow. Two of the most important goals of Intel MPX are to provide this capability at very low performance overhead for newly compiled code, and to provide compatibility mechanisms with legacy software components. A direct benefit Intel MPX provides is hardening software against malicious attacks designed to cause or exploit buffer overruns. This chapter describes the software visible interfaces of this extension.

9.2 INTRODUCTION

Intel MPX is designed to allow a system (i.e., the logical processor(s) and the OS software) to run both Intel MPX enabled software and legacy software (written for processors without Intel MPX). When executing software containing a mixture of Intel MPX-unaware code (legacy code) and Intel MPX-enabled code, the legacy code does not benefit from Intel MPX, but it also does not experience any change in functionality or reduction in performance. The performance of Intel MPX-enabled code running on processors that do not support Intel MPX may be similar to the use of embedding NOPs in the instruction stream.

Intel MPX is designed such that an Intel MPX enabled application can link with, call into, or be called from legacy software (libraries, etc.) while maintaining existing application binary interfaces (ABIs). And in most cases, the benefit of Intel MPX requires minimal changes to the source code at the application programming interfaces (APIs) to legacy library/applications. As described later, Intel MPX associates **bounds** with pointers in a novel manner, and the Intel MPX hardware uses **bounds** to check that the pointer based accesses are suitably constrained. Intel MPX enabled software is not required to uniformly or universally utilize the new hardware capabilities over all memory references. Specifically, programmers can selectively use Intel MPX to protect a subset of pointers.

The code enabled for Intel MPX benefits from memory protection against vulnerability such as buffer overrun. Therefore there is a heightened incentive for software vendors to adopt this technology. At the same time, the security benefit of Intel MPX-protection can be implemented according to the business priorities of software vendors. A software vendor can choose to adopt Intel MPX in some modules to realize partial benefit from Intel MPX quickly, and introduce Intel MPX in other modules in phases (e.g. some programmer intervention might be required at the interface to legacy calls). This adaptive property of Intel MPX is designed to give software vendors control on their schedule and modularity of adoption. It also allows a software vendor to secure defense for higher priority or more attack-prone software first; and allows the use of Intel MPX features in one phase of software engineering (e.g., testing) and not in another (e.g., general release) as dictated by business realities.

The initial goal of Intel MPX is twofold: (1) provide means to defend a system against attacks that originate external to some trust perimeter where the trust perimeter subsumes the system memory and integral data repositories, and (2) provide means to pinpoint accidental logic defects in pointer usage, by undergirding memory references with hardware based pointer validation.

As with any instruction set extensions, Intel MPX can be used by application developers beyond detecting buffer overflow, the processor does not limit the use of Intel MPX for buffer overflow detection.

9.3 INTEL MPX PROGRAMMING MODEL

Intel MPX introduces new **bounds register**s and new instructions that operate on bounds registers. Intel MPX allows an OS to support user mode software (operating at CPL=3) and supervisor mode software (CPL < 3) to add memory protection capability against buffer overrun. It provides controls to enable Intel MPX extensions for user mode and supervisor mode independently. Intel MPX extensions are designed to allow software to associate bounds with pointers, and allow software to check memory references against the bounds associated with the

pointer to prevent out of bound memory access (thus preventing buffer overflow).The bounds registers hold lower bound and upper bound that can be checked when referencing memory. An out-of-bounds memory reference then causes a #BR exception. Intel MPX also introduces configuration facilities that the OS must manage to support enabling of user-mode (and/or supervisor-mode) software operations using bounds registers.

9.3.1 Detection and Enumeration of Intel MPX Interfaces

Detection of hardware support for processor extended state component is provided by the main CPUID leaf function 0DH with index ECX = 0. Specifically, the return value in EDX:EAX of CPUID.(EAX=0DH, ECX=0) provides a 64-bit wide bit vector of hardware support of processor state components.

If CPUID.(EAX=07H, ECX=0H).EBX.MPX [bit 14] = 1 (the processor supports Intel MPX), bits [4:3] of CPUID.(EAX=0DH, ECX=0) enumerates the state components associated with Intel MPX. The two component states of Intel MPX are:

- BNDREGS: CPUID.(EAX=0DH, ECX=0):EAX[3] indicates XCR0.BNDREGS[bit 3] is supported. This bit indicates bound register component of Intel MPX state, comprised of four bounds registers, BND0-BND3 (see Section 9.3.4).
- BNDCSR: CPUID.(EAX=0DH, ECX=0):EAX[4] indicates XCR0.BNDCSR[bit 4] is supported. This bit indicates bounds configuration and status component of Intel MPX comprised of BNDCFGU and BNDSTATUS. OS must enable both BNDCSR and BNDREGS bits in XCR0 to ensure full Intel MPX support to applications.
- The size of the processor state component, enabled by XCR0.BNDREGS, is enumerated by CPUID.(EAX=0DH, ECX=03H).EAX[31:0] and the byte offset of this component relative to the beginning of the XSAVE/XRSTOR area is reported by CPUID.(EAX=0DH, ECX=03H).EBX[31:0].
- The size of the processor state component, enabled by XCR0.BNDCSR, is enumerated by CPUID.(EAX=0DH, ECX=04H).EAX[31:0] and the byte offset of this component relative to the beginning of the XSAVE/XRSTOR area is reported by CPUID.(EAX=0DH, ECX=04H).EBX[31:0].

On processors that support Intel MPX, CPUID.(EAX=0DH, ECX=0):EAX[3] and CPUID.(EAX=0DH, ECX=0):EAX[4] will both be 1. On processors that do not support Intel MPX, CPUID.(EAX=0DH, ECX=0):EAX[3] and CPUID.(EAX=0DH, ECX=0):EAX[4] will both be 0. The layout of XCR0 for extended processor state components defined in Intel Architecture is shown in Figure 9-1.

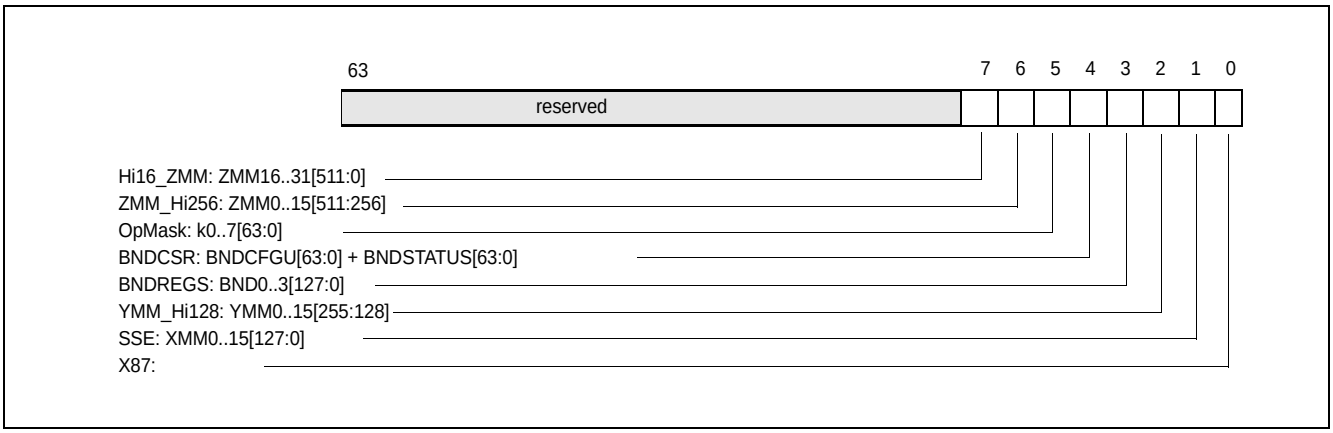


Figure 9-1. Extended Processor State Components defined in Intel Architecture

9.3.2 XSAVE/XRESTOR Support of Intel MPX State

Enabling Intel MPX requires an OS to manage two bits in XCR0 (see bits [4:3] in Table 9-2),

- BNDREGS for saving and restoring BND0-BND3,
- BNDCSR for saving and restoring the user-mode configuration (BNDCFGU) and the status register (BNDSTATUS).

The reason for having two separate bits is that BND0-BND3 is likely to be volatile state, while BNDCFGU and BNDSTATUS are not. Therefore, an OS has flexibility in handling these two states differently in saving or restoring them. The tracking of INIT values is also simplified by using two bits for Intel MPX status (see Table 9-1). The XSAVE/XRSTOR instructions do not support save/restore of IA32_BNDCFGS register (supervisor-mode configuration). An OS must use RDMSR/WRMSR to read/write to IA32_BNDCFGS. XSAVE/XRSTOR is the only interface available to software to access user-mode configuration (BNDCFGU) in both user and supervisor modes.

In both 64-bit and 32-bit/compatibility modes, XSAVE will save the full 128bits of each bounds register (full 64-bit upper bound and full 64-bit lower bound). XRSTOR will sign extend the highest implemented address bit when restoring BNDCFGU and XSAVE will save full 64-bits. 64-bits of BNDSTATUS are saved and restored. No reserved bit checking or canonicality check will be performed on XSAVE/XRSTOR. Intel MPX INIT state is all zeroes for XRSTOR.

The BNDCFGU and BNDSTATUS registers are accessible only with XSAVE/XRSTOR family of instructions. The bounds registers BND0-BND3 can be accessed either via XSAVE/XRSTOR or Intel MPX instructions. Table 9-1 summarizes the behavior of Intel MPX instructions and attempts to modify BND0-BND3, BNDCFGU, BNDSTATUS under different configuration settings of user-mode or supervisor-mode settings and relevant XCR0 settings. Note that while BNDREGS and BNDCSR bits in XCR0 must both be zero or ones to enable Intel MPX instructions, this restriction is does not apply to XSAVE/XRSTOR instructions, allowing these two parts of state to be saved/restored independently

Table 9-1. Intel MPX Feature Enabling

CR4	XCR0		IA32_BNDCFGS	BNDCFGU	Intel MPX Instruction Behavior		Load/Store to BND0-BND3, BNDCFGU, BNDSTATUS	
OXSAVE	BndRegs	BndCSR	Bit 0	Bit 0	CPL0-2	CPL3	XSAVE	XRSTOR
0	NA	NA	NA	NA	NOP	NOP	#UD	#UD
1	0	0	X	X	NOP	NOP	No	No
1	1	1	0	0	NOP	NOP	Yes	Yes
1	1	1	0	1	NOP	MPX	Yes	Yes
1	1	1	1	0	MPX	NOP	Yes	Yes
1	1	1	1	1	MPX	MPX	Yes	Yes

9.3.3 Enabling of Intel MPX States

An OS can enable Intel MPX states to support software operation using bounds registers with the following steps:

- Verify the processor supports XSAVE/XRSTOR/XSETBV/XGETBV instructions and XCR0 by checking CPUID.1.ECX.XSAVE[bit 26]=1.
- Verify the processor supports both Intel MPX states by checking CPUID.(EAX=0DH, ECX=0):EAX[4:3] is 11b.
- Determine the buffer size requirement for the XSAVE area that will be used by XSAVE/XRSTOR (see Section 9.3.1).
- Set CR4.OSXSAVE[bit 18]=1 to enable the use of XSETBV/XGETBV instructions to write/read XCR0.
- Supply an appropriate mask via EDX:EAX to execute XSETBV to enable the processor state components. If XCR0[BNDREGS] != XCR0[BNDCSR], an attempt to execute XSETBV will cause #GP.

Table 9-2. XCR0 Processor State Component Management Controls for Intel MPX

Bit	Meaning
0 - x87	This bit 0 must be 1. An attempt to write 0 to this bit causes a #GP exception.
1 - SSE	If 1, the processor supports SSE state (MXCSR and XMM registers) management using XSAVE, XSAVEOPT, and XRSTOR.
2 - YMM_Hi128	If 1, the processor supports YMM_hi128 state management (upper 128 bits of YMM0-15) using XSAVE, XSAVEOPT, and XRSTOR.
3 - BNDREGS	If 1, the processor supports Intel Memory Protection Extensions (Intel MPX) bounds register state management using XSAVE, XSAVEOPT, and XRSTOR.
4 - BNDCSR	If 1, the processor supports Intel MPX bound configuration and status management using XSAVE, XSAVEOPT, and XRSTOR.
5 - OPMASK	If 1, the processor supports Opmask register state management using XSAVE, XSAVEOPT, and XRSTOR.
6 - ZMM_Hi256	If 1, the processor supports ZMM0-ZMM15 register upper 256-bit state management using XSAVE, XSAVEOPT, and XRSTOR.
7 - Hi16_ZMM	If 1, the processor supports ZMM16-ZMM31 register 512-bit state management using XSAVE, XSAVEOPT, and XRSTOR.

9.3.4 Bounds Registers

Intel MPX Architecture defines four new registers, BND0-BND3, which Intel MPX instructions operate on. Each bounds register stores a pair of 64-bit values which are the lower bound (LB) and upper bound (UB) of a buffer, see Figure 9-2.

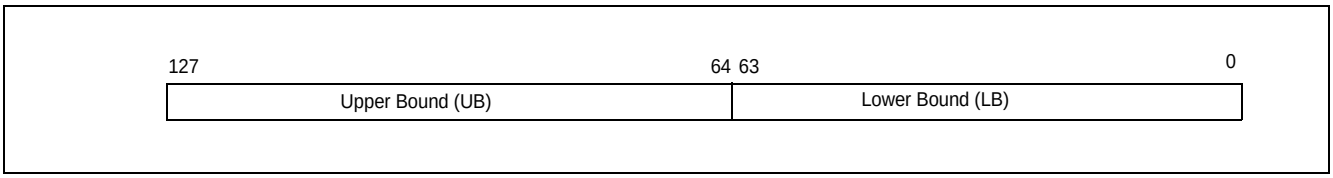


Figure 9-2. Layout of the Bounds Registers BND0-BND3

The bounds are unsigned effective addresses, and are inclusive. The upper bounds are architecturally represented in 1's complement form. Lower bound = 0, and upper bound = 0 (1's complement of all 1s) will allow access to the entire address space. The bounds are considered as INIT when both lower and upper bounds are 0 (cover the entire address space). The two Intel MPX instructions which operate on the upper bound (BNDMK and BNDCU) account for the 1's complement representation of the upper bounds.

The instruction set does not impose any conventions on the use of bounds registers. Software has full flexibility associating pointers to bounds registers including sharing them for multiple pointers.

RESET or INIT# will INIT (write zero) to BND0-BND3

9.3.5 Configuration and Status Registers

Intel MPX defines two configuration and one status registers. The two configuration registers are defined for user mode (CPL 3) and supervisor mode (CPL 0, 1 and 2). The user-mode configuration register BNDCFGU is accessible only with the XSAVE/XRSTOR family of instructions. The supervisor mode configuration register is an architecture MSR, referred to as IA32_BNDCFGS (MSR 0D90H). Because both configuration registers share a common layout (see Figure 9-3), when describing the common behavior, these configuration registers are often denoted as BNDCFGx, where x can be U or S, for user and supervisor mode respectively.

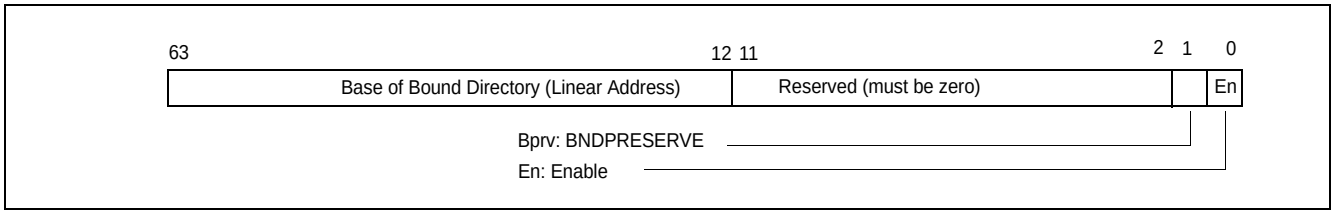


Figure 9-3. Common Layout of the Bound Configuration Registers BNDCFGU and BNDCFGS

The Enable bit in BNDCFGU enables Intel MPX in user mode (see Table 9-2), and the Enable bit in BNDCFGS enables Intel MPX in supervisor mode. The BNDPRESERVE bit controls the initialization behavior of CALL/RET/JMP/Jcc instructions which don't have the BND (0xF2) prefix -- see Section 9.3.12.

The reserved area must be zero for BNDCFGS (WRMSR to BNDCFGS will #GP if the reserved bits of BNDCFGS are not all zeros. XRSTOR of BNDCFGU will not fault if reserved bits are non-zero).

The base of bound directory is a 4K page aligned linear address, and is always in canonical form. Any load into BNDCFGx (XRSTOR or WRMSR) ensures that the highest implemented bit of the linear address is sign extended to guarantee the canonicality of this address.

Intel MPX also defines a status register (BNDSTATUS) primarily used to communicate status information for #BR exception. The layout of the status register is shown in Figure 9-4.

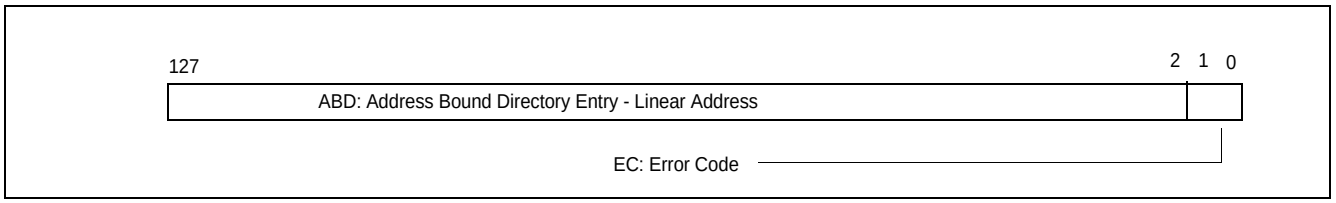


Figure 9-4. Layout of the Bound Status Registers BNDSTATUS

The BNDSTATUS register provides two fields to communicate the status of Intel MPX operations:

- EC (bits 1:0): The error code field communicates status information of a bound range exception #BR or operation involving bound directory.
- ABD: (bits 63:2):The address field of a bound directory entry can provide information when operation on the bound directory caused a #BR.

The valid error codes are defined in Table 9-3.

Table 9-3. Error Code Definition of BNDSTATUS

EC	Description	Meaning
00b[1]	No Intel MPX exception	No exception caused by Intel MPX operations.
01b	Bounds violation	#BR caused by BNDCL, BNDCU or BNDCN instructions; ABD is 0.
10b	Invalid BD entry	#BR caused by BNDLDX or BNDSTX instructions, ABD will be set to the linear address of the invalid Bound directory entry
11b	Reserved	Reserved

NOTES:

1. When legacy BOUND instruction cause a #BR with Intel MPX enabled (see Section 9.3.13), EC is written with Zero.

RESET or INIT# will set BNDCFGx and BNDSTATUS registers to zero.

9.3.5.1 Read and write to IA32_BNDCFGS

The read and write MSR instructions are used to read/write IA32_BNDCFGS (XSAVE state does not include IA32_BNDCFGS). The write MSR instruction to IA32_BNDCFGS checks for canonicality of the addresses being loaded into IA32_BNDCFGS independent of the mode (loads full 64-bit address and performs canonical address check in both 32-bit and 64-bit modes). It will #GP if canonical address reserved bits (must be zero) check fails.

Software can always read/write IA32_BNDCFGS using read/write MSR instruction as long as the processor implements Intel MPX, i.e. CPUID.(EAX=07H, ECX=0H).EBX.MPX = 1. The states of CR4 and XCR0 have no impact on read/write to IA32_BNDCFGS.

9.3.6 Intel MPX Instruction Summary

When Intel MPX is not enabled or not present, all Intel MPX instructions behave as NOP. There are eight Intel MPX instructions, Table 9-4 provides a summary.

A C/C++ compiler can implement intrinsic support for Intel MPX instructions to facilitate pointer operation with capability of checking for valid bounds on pointers. Typically, Intel MPX intrinsics are implemented by compiler via inline code generation where bounds register allocations are handled by the compiler without requiring the programmer to directly manipulate any bounds registers. Therefore no new data type for a bounds register is needed in the syntax of Intel MPX intrinsics.

Table 9-4. Intel MPX Instruction Summary

Intel MPX Instruction	Description
BNDMK b, m	Create LowerBound (LB) and UpperBound (UB) in the bounds register b
BNDCL b, r/m	Checks the address of a memory reference or address in r against the lower bound
BNDCU b, r/m	Checks the address of a memory reference or address in r against the upper bound in 1's complement form
BNDCN b, r/m	Checks the address of a memory reference or address in r against the upper bound not in 1's complement form
BNDMOV b, b/m	Copy/load LB and UB bounds from memory or a bounds register
BNDMOV b/m, b	Store LB and UB bounds in a bounds register to memory or another register
BNDLDX b, mib	Load bounds using address translation using an sib-addressing expression mib
BNDSTX mib, b	Store bounds using address translation using an sib-addressing expression mib

9.3.7 Usage and Examples

BNDMK is typically used after memory is allocated for a buffer, e.g., by functions such as malloc, calloc, or when the memory is allocated on the stack. However, many other usages are possible such as when accessing an array member of a structure.

Example 9-1. BNDMK Example Usage in Application and Library Code

```
int A[100]; //assume the array A is allocated on the stack at 'offset' from RBP.
// the instruction to store starting address of array will be:
        LEA RAX, [RBP+offset]
// the instruction to create the bounds for array A will be:
        BNDMK BND0, [RAX+399]
// Store RAX into BND0.LB, and ~(RAX+399) into BND0.UB.
```

```
// similarly, for a library implementation of dynamic allocated memory
        int * k = malloc(100);
// assuming that malloc returns pointer k in RAX and holds (size - 1) in RCX
// the malloc implementation will execute the following instruction before returning:
        BNDMK BND0, [RAX+RCX]
// BND0.LB stores RAX, and BND0.UB stores ~(RAX+RCX)
```

BNDMOV is typically used to copy bounds from one bound register to another when a pointer is copied from one general purpose register to another, or to spill/fill bounds into memory corresponding to a spill/fill of a pointer.

Example 9-2. BNDMOV Example

Spilling or caller save of bound register would use BNDMOV [RBP+ offset], BNDx. Assuming that the calling convention is that bound of first pointer is passed in BND0, and that bound happens to be in BND3 before the call, the software will add instruction BNDMOV BND0, BND3 prior to the call.

BNDCL/BNDCU/BNDCN are typically used before writing to a buffer but can be used in other instances as well. If there are no bounds violations as a result of bound check instruction, the processor will proceed to execute the next instruction. However, if the bound check fails, it will signal #BR exception (fault).

Typically, the pointer used to write to memory will be compared against lower bound. However, for upper bound check, the software must add the (operand size - 1) to the pointer before upper bound checking.

For example, the software intend to write 32-bit integer in 64-bit mode into a buffer at address specified in RAX, and the bounds are in register BND0, the instruction sequence will be:

```
BNDCL BND0, [RAX]
BNDCU BND0, [RAX+3] ; operand size is 4
MOV Dword ptr [RAX], RBX ; RBX has the data to be written to the buffer.
```

Software may move one of the two bound checks out of a loop if it can determine that memory is accessed strictly in ascending or descending order. For string instructions of the form REP MOVS, the software may choose to do check lower bound against first access and upper bound against last access to memory. However, if software wants to also check for wrap around conditions as part of address computation, it should check for both upper and lower bound for first and last instructions (total of four bound checks).

BNDSTX is used to store the bounds associated with a buffer and the "pointer value" of the pointer to that buffer onto a bound table entry via address translation using a two-level structure, see Section 9.3.8.

For example, the software has a buffer with bounds stored in BND0, the pointer to the buffer is in ESI, the following sequence will store the "pointer value" (the buffer) and the bounds into a configured bound table entry using address translation from the linear address associated with the base of a SIB-addressing form consisting of a base register and a index register:

```
MOV ECX, Dword ptr [ESI] ; store the pointer value in the index register ECX
MOV EAX, ESI ; store the pointer in the base register EAX
BNDSTX Dword ptr [EAX+ECX], BND0 ; perform address translation from the linear address of the base EAX and store bounds and pointer value ECX onto a bound table entry.
```

Similarly to retrieve a buffer and its associated bounds from a bound table entry:

```
MOV EAX, dword ptr [EBX] ;
BNDLDX BND0, dword ptr [EBX+EAX]; perform address translation from the linear address of the base EBX, and loads bounds and pointer value from a bound table entry
```

9.3.8 Loading and Storing Bounds using Translation

Intel MPX defines two instructions for load/store of the linear address of a pointer to a buffer, along with the bounds of the buffer into a paging structure of extended bounds. Specifically when storing extended bounds, the processor will perform address translation of the address where the pointer is stored to an address in the Bound Table (BT) to determine the store location of extended bounds. Loading of an extended bounds performs the reverse sequence.

The structure in memory to load/store an extended bound is a 4-tuple consisting of lower bound, upper bound, pointer value and a reserved field (for use by future versions of Intel MPX, software must not use this field). Bound loads and stores access 32-bit or 64-bit operand size according to the operation mode. Thus, a bound table entry is 4*32 bits in 32-bit mode and 4*64 bits in 64-bit mode. The linear address of a bound table is stored in a Bound Directory (BD) entry. And the linear address of the bound directory is derived from either BNDCFGU or BNDCFGS. Bounds in memory are stored in Bound Tables (BT) as an extended bound, which are accessed via Bound Directory (BD) and address translation performed by BNDLDX/BNDSTX instructions.

Bounds Directory (BD) and Bounds Tables (BT) are stored in application memory and are allocated by the application (in case of kernel use, the structures will be in kernel memory). The bound directory and each instance of bound table are in contiguous linear memory. Figure 9-5 shows the two-level structures for address translation of extended bounds in 64-bit mode. The bound directory contains 8-byte entries and can hold 2^28 entries. The address of the bound directory is located from either BNDCFGx. BNDCFGx contains the linear address in canonical form.

The 64-bit mode address translation mechanism for the two-level structures to access extended bounds consist of:

- A 4-KByte naturally aligned bound directory is located at the linear address specified in bits 63:12 of BNDCFGx (see Figure 9-3). A 64-bit mode bound directory comprises of 2^28 64-bit entries (BDEs). A BDE is selected using the LAp (linear address of pointer to a buffer) to construct an index, comprised of:
  - Bits 30: 3 are from LAp[47:20].
  - Bits 2:0 are 0.

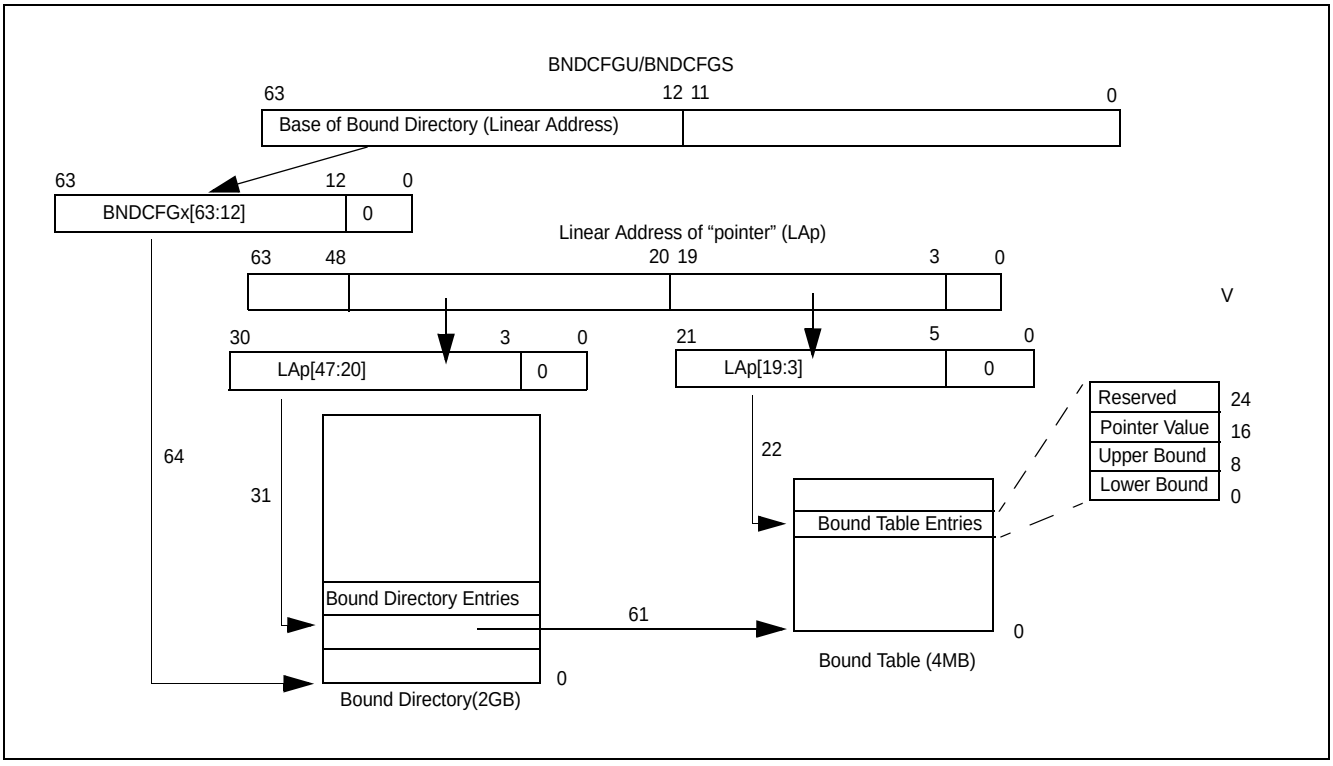


Figure 9-5. Bound Paging Structure and Address Translation in 64-bit Mode

- Each valid BDE contains a valid bit field (bit 0) and a BT address field that points to a bound table. The valid field indicates the BT address field is valid if 1. Each bound table is 8-byte naturally aligned and located at the linear address specified by the BT address field of the BDE. The bound table is located at the linear address of the BT address field shift left by 3 bits for an 8-byte aligned linear address, see Figure 9-6. A 64-bit mode bound table comprises 2^17 bound table entries (BTEs). A BTE is selected using the LAp (linear address of pointer to a buffer) to construct an index, comprised of:
  - Bits 21: 5 are from LAp[19:3].
  - Bits 4:0 are 0.
- Each bound table entry is comprised of
  - the lower bound (LB) field is 64-bit wide
  - the upper bound (UB) field is 64-bit wide
  - the pointer value is 64-bit wide

— reserved field is 64-bit wide, and is reserved for future Intel MPX. Software must not use this field

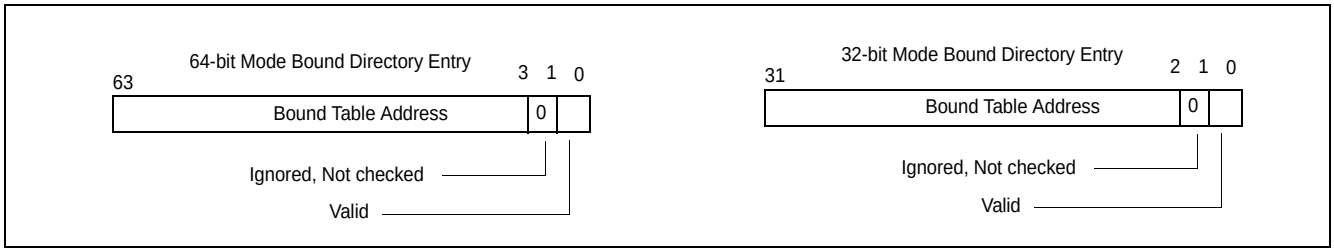


Figure 9-6. Layout of a Bound Directory Entry

Figure 9-6 shows the format of a bound directory entry for 32-bit and 64-bit modes, which comprised of:

- Valid (V, bit 0): entry is not valid if 0, valid if 1;
- The following bits are not used and not checked
  - — 32-bit mode: Bit 1
  - — 64-bit mode: Bits 2 and 1
- BT address field (bits 63:3 for 64-bit mode, bits 31:2 for 32-bit mode) is the address of the bound table pointed by this entry.

The BT address field is valid only if V is 1. If V=0, use of this entry by BNDLDX and BNDSTX will cause #BR and set the error code to 10 and copy bits [63:02] of the address of BD entry into BNDSTATUS register

In 64-bit mode, BT Address field specifies Bits 63-3, and Bits 2-0 of BT address are assumed to be zero. Given that the processor treats segment base of DS as zero in this mode, the BT address specified here is the final address used to access BT

In 32-bit and compatibility mode, BT Address field specifies Bits 31-2, and Bits 1-0 of BT address are assumed to be zero. BT address specifies an effective address in DS segment which is always used in this address calculation.

Limit checking of segment descriptor generally applies to address translation of extended bounds. E.g., when DS is a NULL segment, limit checking will signal #GP in 32-bit but 64-bit mode does not perform limit check.

Figure 9-7 shows the 32-bit mode address translation mechanism for the two-level structures of extended bounds.

The 32-bit mode address translation mechanism for the two-level structures to access extended bounds consist of:

- A 4-KByte naturally aligned bound directory is located at the linear address specified in bits 31:12 of BNDCFGx (see Figure 9-3). A 32-bit mode bound directory comprises of 2^20 32-bit entries (BDEs). A BDE is selected using the LAp (linear address of pointer to a buffer) to construct an index, comprised of:
  - — Bits 21: 2 are from LAp[31:12].
  - — Bits 1:0 are 0.
- Each valid BDE contains a valid bit field (bit 0) and a BT address field that points to a bound table. The valid field indicates the BT address field is valid if 1. Each bound table is 4-byte naturally aligned and located at the linear address specified by the BT address field of the BDE. The bound table is located at the linear address of the BT address field shift left by 2 bits for an 4-byte aligned linear address, see Figure 9-6. A 32-bit mode bound table comprises 2^10 bound table entries (BTEs). A BTE is selected using the LAp (linear address of pointer to a buffer) to construct an index, comprised of:
  - — Bits 13:4 are from LAp[11:3].
  - — Bits 3:0 are 0.

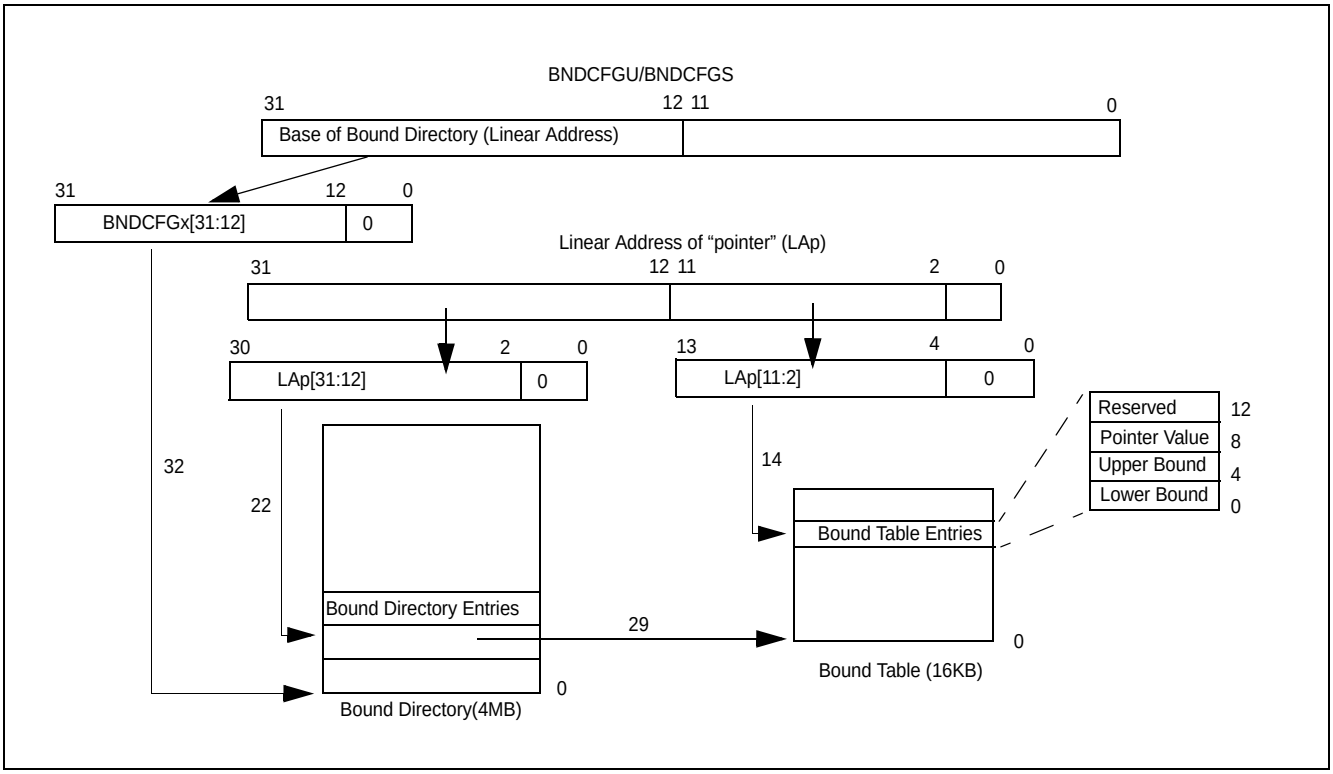


Figure 9-7. Bound Paging Structure and Address Translation in 32-bit Mode

- Each bound table entry is comprised of
  - — the lower bound (LB) field is 32-bit wide
  - — the upper bound (UB) field is 32-bit wide
  - — the pointer value is 32-bit wide
  - — reserved field is 32-bit wide, and is reserved for future Intel MPX. Software must not use this field.

Bounds in memory are associated with the memory address where the pointer is stored, i.e., Ap. Linear address LAp is computed by adding segment base to Ap (note that segment override to these instructions applies to computation of LAp only). The upper 20 bits LAp[31:12] in protected/compatibility modes or upper 28 bits LAp[47:20] in 64-bit mode (IA-32e architecture currently implements 48-bits of virtual address space) are used to index into the bound directory BD. The base address of BD is obtained from BNDCFGx[63:12]. As mentioned in Section 9.3.5, BNDCFGx contains linear address in canonical form. Each valid BD entry points to a bound table BT. In 32-bit and compatibility mode, this is an effective address in DS segment. In 64-bit mode, this is the final address used for BT access because DS segment base is treated as zero by processor. Bits LAp[11:2] in protected/compatibility modes or bits LAp[19:3] in 64-bit mode are used to index into BT. Each entry in BT contains lower bound, upper bound, pointer value and a reserved field.

9.3.9 Instruction Encoding

All Intel MPX instructions are NOP on processors that report CPUID.(EAX=07H, ECX=0H).EBX.MPX [bit 14] = 0, or if Intel MPX is not enabled by the operating system (see Section 9.3.3). Applications can selectively opt-in to use Intel MPX instructions.

All Intel MPX opcodes encoded to operate on BND0-BND3 are valid Intel MPX instructions. All Intel MPX opcodes encoded to operate on bound registers beyond BND3 will #UD if Intel MPX is enabled.

BNDLDX/BNDSTX opcodes require 66H as a mandatory prefix with its operand size tied to the address size attribute of the supported operating modes. Attempt to override operand size attribute with 66H or with REX.W in 64-bit mode is ignored.

9.3.10 Intel MPX and Operating Modes

In 64-bit Mode, all Intel MPX instructions use 64-bit operands for bounds and 64 bit addressing, i.e. REX.W & 67H have no effect on data or address size.

XSAVE, XSAVEOPT and XRSTOR load/store 64-bit values in all modes, as these state-management instructions are not Intel MPX instructions.

In compatibility and legacy modes (including 16-bit code segments, real and virtual 8086 modes) all Intel MPX instructions use 32-bit operands for bounds and 32 bit addressing. The upper 32-bits of destination bound register are cleared (consistent with behavior of integer registers)

In 32-bit and compatibility mode, the bounds are 32-bit, and are treated same as 32-bit integer registers. Therefore, when 32-bit bound is updated in a bound register, the upper 32-bits are undefined. When switching from 64-bit, the behavior of content of bounds register will be similar to that of general purpose registers.

Table 9-5 describes the impact of 67H prefix on memory forms of Intel MPX instructions (register-only forms ignore 67H prefix) when Intel MPX is enabled:

Table 9-5. Effective Address Size of Intel MPX Instructions with 67H Prefix

Addressing Mode	67H Prefix	Effective Address Size used for Intel MPX instructions when Intel MPX is enabled
64-bit Mode	Y	64 bit addressing used
64-bit Mode	N	64 bit addressing used
32-bit Mode	Y	#UD
32-bit Mode	N	32 bit addressing used
16-bit Mode	Y	32 bit addressing used
16-bit Mode	N	#UD

9.3.11 Intel MPX Support for Pointer Operations with Branching

Intel MPX provides flexibility in supporting pointer operation across control flow changes. Intel MPX allows

- compatibility with legacy code that may perform pointer operation across control flow changes and are unaware of Intel MPX, along with
- Intel MPX-aware code that adds bounds checking protection to pointer operation across control flow changes.

The interface to provide such flexibility consists of:

- Using a prefix, referred to as BND prefix, to relevant branch instructions: call, ret, jmp and jcc
- BNDCFGU and BNDCFGS provides the bit field, BNDPRESERVE (bit 1).

The value of BNDPRESERVE in conjunction with the presence/absence the BND prefix with those branching instruction will determine whether the values in BND0-BND3 will be initialized or unchanged.

9.3.12 CALL, RET, JMP and All Jcc

An application compiled to use Intel MPX will use the REPNE (0xF2) prefix (denoted by BND) for all forms of near CALL, near RET, near JMP, short & near Jcc instructions (BND+CALL, BND+RET, BND+JMP, BND+Jcc). See Table 9-6 for specific opcodes. All far CALL, RET and JMP instructions plus short JMP (JMP rel 8, opcode EB) instructions will never cause bound registers to be initialized.

If BNDPRESERVE bit is one, above instructions will NOT INIT the bounds registers when BND prefix is not present for above instructions (legacy behavior). However, If BNDPRESERVE is zero, above instructions will INIT ALL bound

registers (BND0-BND3) when BND prefix is not present for above instructions. If BND prefix is present for above instructions, the BND registers will NOT INIT any bound registers (BND0-BND3).

The legacy code will continue to use non-prefixed forms of these instructions, so if BNDPRESERVE is zero, all the bound registers will INIT by legacy code. This allows the legacy function to execute and return to callee with all bound registers initialized (legacy code by definition cannot make or load bounds in bound registers because it does not have Intel MPX instructions). This will eliminate compatibility concerns when legacy function might have changed the pointer in registers but did not update the value of the bounds registers associated with these pointers.

If BNDCFGx.BNDPRESERVE is clear then non-prefixed forms of these instructions will initialize all the bound registers. If this bit is set then non-prefixed and prefixed forms of these instructions will preserve the contents of bound registers as shown in Table 9-6.

Table 9-6. Bounds Register INIT Behavior Due to BND Prefix with Branch Instructions

Instruction	Branch Instruction Opcodes	BNDPRESERVE = 0	BNDPRESERVE = 1
CALL	E8, FF/2	Init BND0-BND3	BND0-BND3 unchanged
BND + CALL	F2 E8, F2 FF/2	BND0-BND3 unchanged	BND0-BND3 unchanged
RET	C2, C3	Init BND0-BND3	BND0-BND3 unchanged
BND + RET	F2 C2, F2 C3	BND0-BND3 unchanged	BND0-BND3 unchanged
JMP	E9, FF/4	Init BND0-BND3	BND0-BND3 unchanged
BND + JMP	F2 E9, F2 FF/4	BND0-BND3 unchanged	BND0-BND3 unchanged
Jcc	70 through 7F, 0F 80 through 0F 8F	Init BND0-BND3	BND0-BND3 unchanged
BND + Jcc	F2 70 through F2 7F, F2 0F 80 through F2 0F 8F	BND0-BND3 unchanged	BND0-BND3 unchanged

9.3.13 BOUND Instruction and Intel MPX

If Intel MPX in enabled (as specified by Table 9-1) and a #BR was caused due to a BOUND instruction, then BOUND instruction will write zero to the BNDSTATUS register. In all other situations, BOUND instruction will not modify BNDSTATUS. Specifically, the operation of the BOUND instruction can be described as:

```
IF ( ( BOUND instruction caused #BR) AND ( CR4.OXXSAVE =1 AND XCR0.BNDREGS=1 AND XCR0.BNDCSR =1) AND
    ( (CPL=3 AND BNDCFGU.ENABLE = 1) OR (CPL < 3 AND BNDCFGS.ENABLE = 1) ) ) THEN
    BNDSTATUS ← 0;
ELSE
    BNDSTATUS is not modified;
FI;
```

9.3.14 Programming Considerations

Intel MPX instruction set does not dictate any calling convention, but allows the calling convention extensions to be interoperable with legacy code by making use of the of the bound registers and the bound tables to convey arguments and return values.

9.3.15 Intel MPX and System Manage Mode

Upon delivery of an SMI to a processor supporting Intel MPX, the content of IA32_BNDCFGS is saved to SMM state save map and cleared when entering into SMM. RSM will restore IA32_BNDCFGS from the SMM state save map. Offset 7ED0H in SMM state save map will store the content of IA32_BNDCFGS. RSM will load only bits 47:12 and bits 1-0 from SMRAM: bits 11:2 are forced to 0 regardless of what is in SMM state save map; RSM will sign-extend bit 47 into bits 63:48 regardless of what is in SMM state save map.

The content of IA32_BNDCFGS is cleared after entering into SMM. Thus, Intel MPX is disabled inside an SMM handler until SMM code enables it explicitly. This will prevent the side-effect of INIT-ing bound registers by legacy CALL/RET/JMP/Jcc in SMM code.

9.3.16 Support of Intel MPX in VMCS

A new guest-state field for IA32_BNDCFGS is added to the VMCS. In addition, two new controls are added:

- a VM-exit control called "clear BNDCFGS"
- a VM-entry control called "load BNDCFGS."

VM exits always save IA32_BNDCFGS into BNDCFGS field of VMCS; if "clear BNDCFGS" is 1, VM exits clear IA32_BNDCFGS. If "load BNDCFGS" is 1, VM entry loads IA32_BNDCFGS from VMCS. If loading IA32_BNDCFGS, VM entry should check the value of that register in the guest-state area of the VMCS and cause the VM entry to fail (late) if the value is one that would causes WRMSR to fault if executed in ring 0.

9.3.17 Support of Intel MPX in Intel TSX

For some processor implementations, the following Intel MPX instructions may always cause transactional aborts:

- An Intel TSX transaction abort will occur in case of legacy branch (that causes bounds registers INIT) when at least one bounds register was in a NON-INIT state.
- An Intel TSX transaction abort will occur in case of a BNDLDX & BNDSTX instruction on non-flat segment.

Intel MPX Instructions (including BND prefix + branch instructions) not enumerated above as causing transactional abort when used inside a transaction will typically not cause an Intel TSX transaction to abort.

9.4 INTEL MPX INSTRUCTION REFERENCE

9.4.1 Instruction Column in the Instruction Summary Table

- **bnd** — a 128-bit bounds register. BND0 through BND3.
- **mib** — a memory operand using SIB addressing form, where the index register is not used in address calculation, Scale is ignored. Only the base and displacement are used in effective address calculation.

BNDMK—Make Bounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1B /r BNDMK bnd, m32	RM	NE/V	MPX	Make lower and upper bounds from m32 and store them in bnd.
F3 0F 1B /r BNDMK bnd, m64	RM	V/NE	MPX	Make lower and upper bounds from m64 and store them in bnd.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	ModRM:r/m (r)	NA

Description

Makes bounds from the second operand and stores the lower and upper bounds in the bound register bnd. The second operand must be a memory operand. The content of the base register from the memory operand is stored in the lower bound bnd.LB. The 1's complement of the effective address of m32/m64 is stored in the upper bound b.UB. Computation of m32/m64 has identical behavior to LEA.

This instruction does not cause any memory access, and does not read or write any flags.

If the instruction did not specify base register, the lower bound will be zero. The reg-reg form of this instruction retains legacy behavior (NOP).

RIP relative instruction in 64-bit will #UD.

Operation

```
BND.LB ← SRCMEM.base;
IF 64-bit mode Then
    BND.UB ← NOT(LEA.64_bits(SRCMEM));
ELSE
    BND.UB ← Zero_Extend.64_bits(NOT(LEA.32_bits(SRCMEM)));
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDMKvoid * _bnd_set_ptr_bounds(const void * q, size_t size);

Flags Affected

None

Protected Mode Exceptions

#UD	If ModRM is RIP relative.
	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions

#UD	If ModRM is RIP relative.
	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Virtual-8086 Mode Exceptions

#UD	If ModRM is RIP relative.
	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#SS(0)	If the memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.

Same exceptions as in protected mode.

BNDCL—Check Lower Bound

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1A /r BNDCL bnd, r/m32	RM	NE/V	MPX	Generate a #BR if the address in r/m32 is lower than the lower bound in bnd.LB.
F3 0F 1A /r BNDCL bnd, r/m64	RM	V/NE	MPX	Generate a #BR if the address in r/m64 is lower than the lower bound in bnd.LB.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	ModRM:r/m (r)	NA

Description

Compare the address in the second operand with the lower bound in bnd. The second operand can be either a register or memory operand. If the address is lower than the lower bound in bnd.LB, it will set BNDSTATUS to 01H and signal a #BR exception.

This instruction does not cause any memory access, and does not read or write any flags.

Operation

BNDCL BND, reg

```
IF reg < BND.LB Then
    BNDSTATUS ← 01H;
    #BR;
FI;
```

BNDCL BND, mem

```
TEMP ← LEA(mem);
IF TEMP < BND.LB Then
    BNDSTATUS ← 01H;
    #BR;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDCL void _bnd_chk_ptr_lbounds(const void *q)

Flags Affected

None

Protected Mode Exceptions

#BR	If lower bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions

#BR	If lower bound check fails.
#UD	If the LOCK prefix is used.

	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Virtual-8086 Mode Exceptions

#BR	If lower bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.

Same exceptions as in protected mode.

BNDCU/BNDCN—Check Upper Bound

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 1A /r BNDCU bnd, r/m32	RM	NE/V	MPX	Generate a #BR if the address in r/m32 is higher than the upper bound in bnd.UB (bnb.UB in 1's complement form).
F2 0F 1A /r BNDCU bnd, r/m64	RM	V/NE	MPX	Generate a #BR if the address in r/m64 is higher than the upper bound in bnd.UB (bnb.UB in 1's complement form).
F2 0F 1B /r BNDCN bnd, r/m32	RM	NE/V	MPX	Generate a #BR if the address in r/m32 is higher than the upper bound in bnd.UB (bnb.UB not in 1's complement form).
F2 0F 1B /r BNDCN bnd, r/m64	RM	V/NE	MPX	Generate a #BR if the address in r/m64 is higher than the upper bound in bnd.UB (bnb.UB not in 1's complement form).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	ModRM:r/m (r)	NA

Description

Compare the address in the second operand with the upper bound in bnd. The second operand can be either a register or a memory operand. If the address is higher than the upper bound in bnd.UB, it will set BNDSTATUS to 01H and signal a #BR exception.

BNDCU perform 1's complement operation on the upper bound of bnd first before proceeding with address comparison. BNDCN perform address comparison directly using the upper bound in bnd that is already reverted out of 1's complement form.

This instruction does not cause any memory access, and does not read or write any flags.

Effective address computation of m32/64 has identical behavior to LEA

Operation

BNDCU BND, reg

```
IF reg > NOT( BND.UB) Then
    BNDSTATUS ← 01H;
    #BR;
FI;
```

BNDCU BND, mem

```
TEMP ← LEA(mem);
IF TEMP > NOT( BND.UB) Then
    BNDSTATUS ← 01H;
    #BR;
FI;
```

BNDCN BND, reg

```
IF reg > BND.UB Then
    BNDSTATUS ← 01H;
    #BR;
FI;
```

BNDCN BND, mem

```
TEMP ← LEA(mem);
IF TEMP > BND.UB Then
    BNDSTATUS ← 01H;
    #BR;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDCU .void _bnd_chk_ptr_ubounds(const void *q)

Flags Affected

None

Protected Mode Exceptions

#BR	If upper bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions

#BR	If upper bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Virtual-8086 Mode Exceptions

#BR	If upper bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.

Same exceptions as in protected mode.

BNDMOV—Move Bounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 1A /r BNDMOV bnd1, bnd2/m64	RM	NE/V	MPX	Move lower and upper bound from bnd2/m64 to bound register bnd1.
66 0F 1A /r BNDMOV bnd1, bnd2/m128	RM	V/NE	MPX	Move lower and upper bound from bnd2/m128 to bound register bnd1.
66 0F 1B /r BNDMOV bnd1/m64, bnd2	MR	NE/V	MPX	Move lower and upper bound from bnd2 to bnd1/m64.
66 0F 1B /r BNDMOV bnd1/m128, bnd2	MR	V/NE	MPX	Move lower and upper bound from bnd2 to bound register bnd1/m128.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA

Description

BNDMOV moves a pair of lower and upper bound values from the source operand (the second operand) to the destination (the first operand). Each operation is 128-bit move. The exceptions are same as the MOV instruction. The memory format for loading/store bounds in 64-bit mode is shown in Figure 9-8.

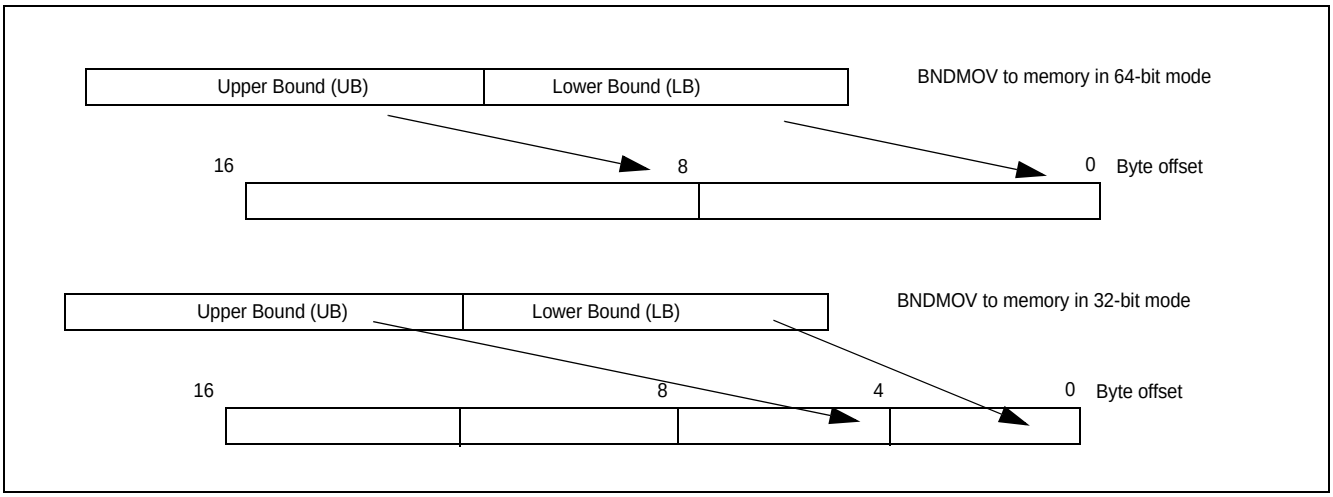


Figure 9-8. Memory Layout of BNDMOV to/from Memory

This instruction does not change flags.

Operation

BNDMOV register to register

DEST.LB ← SRC.LB;
DEST.UB ← SRC.UB;

BNDMOV from memory

```
IF 64-bit mode THEN
        DEST.LB ← LOAD_QWORD(SRC);
        DEST.UB ← LOAD_QWORD( SRC+8);
    ELSE
        DEST.LB ← LOAD_DWORD_ZERO_EXT(SRC);
        DEST.UB ← LOAD_DWORD_ZERO_EXT( SRC+4);
FI;
```

BNDMOV to memory

```
IF 64-bit mode THEN
        DEST[63:0] ← SRC.LB;
        DEST[127:64] ← SRC.UB;
    ELSE
        DEST[31:0] ← SRC.LB;
        DEST[63:32] ← SRC.UB;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDMOV void * _bnd_copy_ptr_bounds(const void *q, const void *r)

Flags Affected

None

Protected Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 67H prefix is not used and CS.D=0. If 67H prefix is used and CS.D=1.
#SS(0)	If the memory operand effective address is outside the SS segment limit.
#GP(0)	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the destination operand points to a non-writable segment If the DS, ES, FS, or GS segment register contains a NULL segment selector.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If the memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If the memory operand effective address is outside the SS segment limit.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand. If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#SS(0)	If the memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code)	If a page fault occurs.

BNDLDX—Load Extended Bounds Using Address Translation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 1A /r BNDLDX bnd, mib	RM	V/V	MPX	Load the bounds stored in a bound table entry (BTE) into bnd with address translation using the base of mib and conditional on the index of mib matching the pointer value in the BTE.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	SIB.base (r): Address of pointer SIB.index(r)	NA

Description

BNDLDX uses the linear address constructed from the base register and displacement of the SIB-addressing form of the memory operand (mib) to perform address translation to access a bound table entry and conditionally load the bounds in the BTE to the destination. The destination register is updated with the bounds in the BTE, if the content of the index register of mib matches the pointer value stored in the BTE.

If the pointer value comparison fails, the destination is updated with INIT bounds (lb = 0x0, ub = 0x0) (note: as articulated earlier, the upper bound is represented using 1's complement, therefore, the 0x0 value of upper bound allows for access to full memory).

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

Operation

```
base ← mib.SIB.base ? mib.SIB.base + Disp: 0;
ptr_value ← mib.SIB.index ? mib.SIB.index : 0;

32-bit protected mod or compatibility mode
A_BDE[31:0] ← (Zero_extend32(base[31:12] << 2) + (BNDCFG[31:12] <<12 );
A_BT[31:0] ← LoadFrom(A_BDE );
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_BTE[31:0] ← (Zero_extend32(base[11:2] << 4) + (A_BT[31:2] << 2 );
Temp_lb[31:0] ← LoadFrom(A_BTE );
Temp_ub[31:0] ← LoadFrom(A_BTE + 4);
Temp_ptr[31:0] ← LoadFrom(A_BTE + 8);
IF Temp_ptr equal ptr_value Then
    BND.LB ← Temp_lb;
    BND.UB ← Temp_ub;
```

```
ELSE
    BND.LB ← 0;
    BND.UB ← 0;
FI;
```

64-bit mode

```
A_BDE[63:0] ← (Zero_extend64(base[47:20] << 3) + (BNDCFG[63:20] <<12 );
A_BT[63:0] ← LoadFrom(A_BDE );
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_BTE[63:0] ← (Zero_extend64(base[19:3] << 5) + (A_BT[63:3] << 3 );
Temp_lb[63:0] ← LoadFrom(A_BTE );
Temp_ub[63:0] ← LoadFrom(A_BTE + 8);
Temp_ptr[63:0] ← LoadFrom(A_BTE + 16);
IF Temp_ptr equal ptr_value Then
    BND.LB ← Temp_lb;
    BND.UB ← Temp_ub;
ELSE
    BND.LB ← 0;
    BND.UB ← 0;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDLDX: Generated by compiler as needed.

Flags Affected

None

Protected Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
	If DS register contains a NULL segment selector.
#PF(fault code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
#PF(fault code)	If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If ModRM is RIP relative.
	If the LOCK prefix is used.
	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#GP(0)	If the memory address (A_BDE or A_BTE) is in a non-canonical form.
#PF(fault code)	If a page fault occurs.

BNDSTX—Store Extended Bounds Using Address Translation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 1B /r BNDSTX mib, bnd	MR	V/V	MPX	Store the bounds in bnd and the pointer value in the index register of mib to a bound table entry (BTE) with address translation using the base of mib.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
MR	SIB.base (r): Address of pointer SIB.index(r)	ModRM:reg (r)	NA

Description

BNDSTX uses the linear address constructed from the displacement and base register of the SIB-addressing form of the memory operand (mib) to perform address translation to store to a bound table entry. The bounds in the source operand bnd are written to the lower and upper bounds in the BTE. The content of the index register of mib is written to the pointer value field in the BTE.

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

Operation

```
base ← mib.SIB.base ? mib.SIB.base + Disp: 0;
ptr_value ← mib.SIB.index ? mib.SIB.index : 0;

32-bit protected mod or compatibility mode
A_BDE[31:0] ← (Zero_extend32(base[31:12] << 2) + (BNDCFG[31:12] <<12 );
A_BT[31:0] ← LoadFrom(A_BDE );
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_DEST[31:0] ← (Zero_extend32(base[11:2] << 4) + (A_BT[31:2] << 2 ); // address of Bound table entry
A_DEST[8][31:0] ← ptr_value;
A_DEST[0][31:0] ← BND.LB;
A_DEST[4][31:0] ← BND.UB;
```

64-bit mode

```
A_BDE[63:0] ← (Zero_extend64(base[47:20] << 3) + (BNDCFG[63:20] <<12 );
A_BT[63:0] ← LoadFrom(A_BDE );
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_DEST[63:0] ← (Zero_extend64(base[19:3] << 5) + (A_BT[63:3] << 3 ); // address of Bound table entry
A_DEST[16][63:0] ← ptr_value;
A_DEST[0][63:0] ← BND.LB;
A_DEST[8][63:0] ← BND.UB;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDSTX: _bnd_store_ptr_bounds(const void **ptr_addr, const void *ptr_val);

Flags Affected

None

Protected Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
	If DS register contains a NULL segment selector.
	If the destination operand points to a non-writable segment
#PF(fault code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
#PF(fault code)	If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If ModRM is RIP relative.
	If the LOCK prefix is used.

	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#GP(0)	If the memory address (A_BDE or A_BTE) is in a non-canonical form.
	If the destination operand points to a non-writable segment
#PF(fault code)	If a page fault occurs.

9.5 INTEL MEMORY PROTECTION EXTENSIONS MSRS

Table 9-7. IA-32 Architectural MSRs for Intel Memory Protection Extensions

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Introduced as Architectural MSR
Hex	**Decimal**			
D90H	3472	IA32_BNDCFGS	**Supervisor Mode Bounds Configuration Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX.[bit 14] = 1)
		0	Enable: Enable Intel MPX for CPL < 3	
		1	BNDPRRESERVE: see Section 9.3.12	
		11:2	Reserved	
		63:12	Linear address of bounds directory	

This page was intentionally left blank.

CHAPTER 10
ADDITIONAL NEW INSTRUCTIONS

This chapter describes additional new instructions for future Intel 64 processors that provide enhancements in selected application domains: ranging from random number generation to multi-precision arithmetic.

10.1 DETECTION OF NEW INTEL® INSTRUCTIONS

Hardware support for flag-preserving add-carry instructions is indicated by the following feature flags:

- CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19]=1 indicates the processor supports ADCX and ADOX instructions.
- Hardware support for the RDSEED instruction is indicated by CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18].
- CPUID.(EAX=8000_0001H):ECX[bit 8]=1 indicates PREFETCHW is supported.

10.2 RANDOM NUMBER INSTRUCTIONS

The instructions for generating random numbers to comply with NIST SP800-90A, SP800-90B, and SP800-90C standards are described in this section.

10.2.1 RDRAND

The RDRAND instruction returns a random number, and was introduced first in the 3rd generation Intel Core processors. All Intel processors that support the RDRAND instruction indicate the availability of the RDRAND instruction via reporting CPUID.01H:ECX.RDRAND[bit 30] = 1.

RDRAND returns random numbers that are supplied by a cryptographically secure, deterministic random bit generator (DRBG). The DRBG is designed to meet the NIST SP 800-90A standard. The DRBG is re-seeded frequently from a on-chip non-deterministic entropy source to guarantee data returned by RDRAND is statistically uniform, non-periodic and non-deterministic.

In order for the hardware design to meet its security goals, the random number generator continuously tests itself and the random data it is generating. Runtime failures in the random number generator circuitry or statistically anomalous data occurring by chance will be detected by the self test hardware and flag the resulting data as being bad. In such extremely rare cases, the RDRAND instruction will return no data instead of bad data.

Under heavy load, with multiple cores executing RDRAND in parallel, it is possible, though unlikely, for the demand of random numbers by software processes/threads to exceed the rate at which the random number generator hardware can supply them. This will lead to the RDRAND instruction returning no data transitorily. The RDRAND instruction indicates the occurrence of this rare situation by clearing the CF flag.

The RDRAND instruction returns with the carry flag set (CF = 1) to indicate valid data is returned. It is recommended that software using the RDRAND instruction to get random numbers retry for a limited number of iterations while RDRAND returns CF=0 and complete when valid data is returned, indicated with CF=1. This will deal with transitory underflows. A retry limit should be employed to prevent a hard failure in the RNG (expected to be extremely rare) leading to a busy loop in software.

The intrinsic primitive for RDRAND is defined to address software's need for the common cases (CF = 1) and the rare situations (CF = 0). The intrinsic primitive returns a value that reflects the value of the carry flag returned by the underlying RDRAND instruction. The example below illustrates the recommended usage of an RDRAND instrinsic in a utility function, a loop to fetch a 64-bit random value with a retry count limit of 10. A C implementation might be written as follows:

```
-------------------------------------------------------------------------------
#define SUCCESS 1
#define RETRY_LIMIT_EXCEEDED 0
#define RETRY_LIMIT 10


int get_random_64( unsigned __int 64 * arand)
{int i     ;
    for ( i = 0; i < RETRY_LIMIT; i ++) {
        if(_rdrand64_step(arand) ) return SUCCESS;
    }
    return RETRY_LIMIT_EXCEEDED;
}
-------------------------------------------------------------------------------
```

The RDRAND instruction is first introduced in third-generation Intel Core processors built on 22-nm process.

10.2.2 RDSEED

The RDSEED instruction returns a random number. All Intel processors that support the RDSEED instruction indicate the availability of the RDSEED instruction via reporting CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 1.

RDSEED returns random numbers that are supplied by a cryptographically secure, enhanced non-deterministic random bit generator (Enhanced NRBG). The NRBG is designed to meet the NIST SP 800-90B and NIST SP800-90C standards.

In order for the hardware design to meet its security goals, the random number generator continuously tests itself and the random data it is generating. Runtime failures in the random number generator circuitry or statistically anomalous data occurring by chance will be detected by the self test hardware and flag the resulting data as being bad. In such extremely rare cases, the RDSEED instruction will return no data instead of bad data.

Under heavy load, with multiple cores executing RDSEED in parallel, it is possible for the demand of random numbers by software processes/threads to exceed the rate at which the random number generator hardware can supply them. This will lead to the RDSEED instruction returning no data transitorily. The RDSEED instruction indicates the occurrence of this situation by clearing the CF flag.

The RDSEED instruction returns with the carry flag set (CF = 1) to indicate valid data is returned. It is recommended that software using the RDSEED instruction to get random numbers retry for a limited number of iterations while RDSEED returns CF=0 and complete when valid data is returned, indicated with CF=1. This will deal with transitory underflows. A retry limit should be employed to prevent a hard failure in the NRBG (expected to be extremely rare) leading to a busy loop in software.

The intrinsic primitive for RDSEED is defined to address software's need for the common cases (CF = 1) and the rare situations (CF = 0). The intrinsic primitive returns a value that reflects the value of the carry flag returned by the underlying RDRAND instruction. The example below illustrates the recommended usage of an RDRAND instrinsic in a utility function, a loop to fetch a 64 bit random value with a retry count limit of 10.

10.2.3 RDSEED and VMX interactions

A VM-execution control exists that allows the virtual machine monitor to trap on the instruction. The "RDSEED exiting" VM-execution control is located at bit 16 of the secondary processor-based VM-execution controls. A VM exit due to RDSEED will have exit reason 61 (decimal).

10.3 PAGING-MODE ACCESS ENHANCEMENT

Intel 64 architecture provided two paging mode modifiers that regulate instruction fetches from linear memory address spaces: Execute-Disable (XD) and supervisor mode execution prevention (SMEP) are described in Chapter 4 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

A third paging mode modifier is introduced to regulate data accesses, referred to as supervisor mode access prevention (SMAP).

When SMAP is enabled, the processor disallows supervisor data accesses to pages that are accessible in user mode. Disallowed accesses result in page-fault exceptions. Software may find it necessary to allow certain supervisor accesses to user-accessible pages. For this reason, the SMAP architecture allows software to disable the SMAP protections temporarily.

SMAP applies in all paging modes (32-bit paging, PAE paging, and IA-32e paging) and to all page sizes (4-KByte, 2-MByte, 4-MByte, and 1-GByte). SMAP has no effect on processor operation if paging is disabled. SMAP has no effect on the address translation provided by EPT. SMAP has no effect in operating modes that paging is not used.

10.3.1 Enumeration and Enabling

System software enables SMAP by setting the SMAP flag in control register CR4 (bit 21).

Processor support for SMAP is enumerated by the CPUID instruction. Specifically, the processor supports SMAP only if CPUID.(EAX=07H,ECX=0H):EBX.SMAP[bit 20] = 1.

A processor will allow CR4.SMAP to be set only if SMAP is enumerated by CPUID as described above. CR4.SMAP may be set if paging is disabled (if CR0.PG = 0), but it has no effect on processor operation in that case.

In addition, two new instructions: CLAC and STAC (see Section 10.6) are supported if and only if SMAP is enumerated by CPUID as described above.

10.3.2 SMAP and Access Rights

Every access to a linear address is either a **supervisor-mode access** or a **user-mode access**. For ordinary accesses, the distinction is based on the current privilege level (CPL): if CPL < 3, accesses are supervisor-mode; if CPL = 3, accesses are user-mode. Some operations implicitly access system data structures, and the resulting accesses to those data structures are supervisor-mode regardless of CPL; these are implicit supervisor accesses[1].

If CR4.SMAP = 1, supervisor-mode data accesses are not allowed to linear addresses that are accessible in user mode. Software can override this protection for ordinary supervisor-mode data accesses by setting EFLAGS.AC; this override does not apply to implicit supervisor-mode accesses.

The following items detail how when supervisor-mode data accesses are allowed:

- Data reads:
  - If CR4.SMAP = 0, supervisor-mode read accesses are allowed from any linear address with a valid translation.
  - If CR4.SMAP = 1, access rights depend on the nature of the access and EFLAGS.AC.
    - An implicit supervisor-mode read access is allowed from a linear address only if the address has a valid translation and the U/S flag (bit 2) is 0 in at least one of the paging-structure entries controlling the translation.
    - If EFLAGS.AC = 0, an ordinary supervisor-mode read access is allowed from a linear address only if the address has a valid translation and the U/S flag is 0 in at least one of the paging-structure entries controlling the translation.
    - If EFLAGS.AC = 1, ordinary supervisor-mode read accesses are allowed from any linear address with a valid translation.

1. Examples of such implicit supervisor accesses include the following: accesses to the global descriptor table (GDT) or local descriptor table (LDT) to load a segment descriptor; accesses to the interrupt descriptor table (IDT) when delivering an interrupt or exception; and accesses to the task-state segment (TSS) as part of a task switch or change of CPL

- Data writes:
  - If CR0.WP = 0 and CR4.SMAP = 0, supervisor-mode write accesses are allowed to any linear address with a valid translation.
  - If CR0.WP = 0 and CR4.SMAP = 1, access rights depend on the nature of the access and EFLAGS.AC:
    - An implicit supervisor-mode write access is allowed to a linear address only if the address has a valid translation and the U/S flag is 0 in at least one of the paging-structure entries controlling the translation.
    - If EFLAGS.AC = 0, an ordinary supervisor-mode write access is allowed to a linear address only if the address has a valid translation and the U/S flag is 0 in at least one of the paging-structure entries controlling the translation.
    - If EFLAGS.AC = 1, ordinary supervisor-mode write accesses are allowed to any linear address with a valid translation.
  - If CR0.WP = 1 and CR4.SMAP = 0, supervisor-mode write accesses are allowed to any linear address with a valid translation for which the R/W flag (bit 1) is 1 in every paging-structure entry.
  - If CR0.WP = 1 and CR4.SMAP = 1, access rights depend on the nature of the access and EFLAGS.AC.
    - An implicit supervisor-mode write access is allowed to a linear address only if the address has a valid translation, the U/S flag is 0 in at least one of the paging-structure entries controlling the translation, and the R/W flag is 1 in every paging-structure entry controlling the translation.
    - If EFLAGS.AC = 0, an ordinary supervisor-mode write access is allowed to a linear address only if the address has a valid translation, the U/S flag is 0 in at least one of the paging-structure entries controlling the translation, and the R/W flag is 1 in every paging-structure entry controlling the translation.
    - If EFLAGS.AC = 1, ordinary supervisor-mode write accesses are allowed to any linear address with a valid translation for which the R/W flag is 1 in every paging-structure entry.

Supervisor-mode data accesses that are not allowed by SMAP cause page-fault exceptions (see Section 4). SMAP has no effect on instruction fetches, user-mode data accesses, or supervisor-mode data accesses to supervisor-only pages.

Software can temporarily disable SMAP for ordinary supervisor-mode accesses by setting EFLAGS.AC. The specifics are described by CLAC and STAC instructions.

10.3.3 SMAP and Page-Fault Exceptions

If SMAP prevents a supervisor-mode access to a linear address, a page-fault exception (#PF) occurs.

SMAP does not define any new bits in the error code delivered by page-fault exceptions. Page-fault exceptions induced by SMAP set the existing bits in the error code as follows:

- P flag (bit 0).

  SMAP causes a page-fault exception only if there is a valid translation for the linear address. Bit 0 of the error code is 1 if there is a valid translation. Thus, page-fault exceptions caused by SMAP always set bit 0 of the error code.

- W/R (bit 1).

  If the access causing the page-fault exception was a write, this flag is 1; otherwise, it is 0.

- U/S (bit 2).

  SMAP causes page-fault exceptions only for supervisor-mode accesses. Bit 2 of the error code is 0 for supervisor-mode accesses. Thus, page-fault exceptions caused by SMAP always clear bit 2 of the error code.

- RSVD flag (bit 3).

  SMAP causes a page-fault exception only if there is a valid translation for the linear address. Bit 3 of the error code is 0 if there is a valid translation. Thus, page-fault exceptions caused by SMAP always clear bit 3 of the error code.

- I/D flag (bit 4).

SMAP causes page-fault exceptions only for data accesses. Bit 4 of the error code is 0 for data accesses. Thus, page-fault exceptions caused by SMAP always clear bit 4 of the error code.

The above items imply that the error code delivered by a page-fault exception due to SMAP is either 1 (for reads) or 3 (for writes). Note that the only page-fault exceptions that deliver an error code of 1 are those induced by SMAP. (If CR0.WP = 1, some page-fault exceptions may deliver an error code of 3 even if CR4.SMAP = 0.)

10.3.4 CR4.SMAP and Cached Translation Information

The MOV to CR4 instruction is not required to invalidate the TLBs or paging-structure caches because of changes being made to CR4.SMAP. If PAE paging is in use, the MOV to CR4 instruction does not cause the PDPTE registers to be reloaded because of changes being made to CR4.SMAP.

10.4 INSTRUCTION EXCEPTION SPECIFICATION

To use this reference of instruction exceptions, look at each instruction for a description of the particular exception type of interest. The instruction's corresponding CPUID feature flag can be identified in the fourth column of the instruction summary table.

Note: #UD on CPUID feature flags=0 is not guaranteed in a virtualized environment if the hardware supports the feature flag.

Table 10-1 lists exception conditions for ADCX and ADOX.

Table 10-1. Exception Definition (ADCX and ADOX Instructions)

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If ADX CPUID feature flag is '0'.
	X	X	X	X	If a LOCK prefix is present.
Stack, SS(0)	X	X	X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

10.5 INSTRUCTION FORMAT

The format used for describing each instruction as in the example below is described in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

ADCX — Unsigned Integer Addition of Two Operands with Carry Flag (THIS IS AN EXAMPLE)

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
66 0F 38 F6 /r ADCX r32, r/m32	RM	V/V	ADX	Unsigned addition of r32 with CF, r/m32 to r32, writes CF.
REX.w + 66 0F 38 F6 /r ADCX r64, r/m64	RM	V/NE	ADX	Unsigned addition of r64 with CF, r/m64 to r64, writes CF.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

10.6 INSTRUCTION SET REFERENCE

ADCX — Unsigned Integer Addition of Two Operands with Carry Flag

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
66 0F 38 F6 /r ADCX r32, r/m32	RM	V/V	ADX	Unsigned addition of r32 with CF, r/m32 to r32, writes CF.
REX.w + 66 0F 38 F6 /r ADCX r64, r/m64	RM	V/NE	ADX	Unsigned addition of r64 with CF, r/m64 to r64, writes CF.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the carry-flag (CF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of CF can represent a carry from a previous addition. The instruction sets the CF flag with the carry generated by the unsigned addition of the operands.

The ADCX instruction is executed in the context of multi-precision addition, where we add a series of operands with a carry-chain. At the beginning of a chain of additions, we need to make sure the CF is in a desired initial state. Often, this initial state needs to be 0, which can be achieved with an instruction to zero the CF (e.g. XOR).

This instruction is supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode.

In 64-bit mode, the default operation size is 32 bits. Using a REX Prefix in the form of REX.R permits access to additional registers (R8-15). Using REX Prefix in the form of REX.W promotes operation to 64 bits.

ADCX executes normally either inside or outside a transaction region.

Note: ADCX defines the OF flag differently than the ADD/ADC instructions as defined in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

Operation

```
IF OperandSize is 64-bit
    THEN CF:DEST[63:0] ← DEST[63:0] + SRC[63:0] + CF;
    ELSE CF:DEST[31:0] ← DEST[31:0] + SRC[31:0] + CF;
FI;
```

Flags Affected

CF is updated based on result. OF, SF, ZF, AF and PF flags are unmodified.

Intel C/C++ Compiler Intrinsic Equivalent

unsigned char _addcarryx_u32 (unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *sum_out);

unsigned char _addcarryx_u64 (unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 10-1.

ADOX — Unsigned Integer Addition of Two Operands with Overflow Flag

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
F3 0F 38 F6 /r ADOX r32, r/m32	RM	V/V	ADX	Unsigned addition of r32 with OF, r/m32 to r32, writes OF.
REX.w + F3 0F 38 F6 /r ADOX r64, r/m64	RM	V/NE	ADX	Unsigned addition of r64 with OF, r/m64 to r64, writes OF.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the overflow-flag (OF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of OF represents a carry from a previous addition. The instruction sets the OF flag with the carry generated by the unsigned addition of the operands.

The ADOX instruction is executed in the context of multi-precision addition, where we add a series of operands with a carry-chain. At the beginning of a chain of additions, we execute an instruction to zero the OF (e.g. XOR).

This instruction is supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode.

In 64-bit mode, the default operation size is 32 bits. Using a REX Prefix in the form of REX.R permits access to additional registers (R8-15). Using REX Prefix in the form of REX.W promotes operation to 64-bits.

ADOX executes normally either inside or outside a transaction region.

Note: ADOX defines the CF and OF flags differently than the ADD/ADC instructions as defined in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

Operation

```
IF OperandSize is 64-bit
    THEN OF:DEST[63:0] ← DEST[63:0] + SRC[63:0] + OF;
    ELSE OF:DEST[31:0] ← DEST[31:0] + SRC[31:0] + OF;
FI;
```

Flags Affected

OF is updated based on result. CF, SF, ZF, AF and PF flags are unmodified.

Intel C/C++ Compiler Intrinsic Equivalent

unsigned char _addcarryx_u32 (unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *sum_out);

unsigned char _addcarryx_u64 (unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 10-1.

PREFETCHW—Prefetch Data into Caches in Anticipation of a Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 0D /1 PREFETCHW m8	M	V/V	PRFCHW	Move data from m8 closer to the processor in anticipation of a write.

Instruction Operand Encoding

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

Description

Fetches the cache line of data from memory that contains the byte specified with the source operand to a location in the 1st or 2nd level cache and invalidates all other cached instances of the line.

The source operand is a byte memory location. (Use of any ModR/M value other than a memory operand will lead to unpredictable behavior.) If the line selected is already present in the lowest level cache and is already in an exclusively owned state, no data movement occurs. Prefetches from non-writeback memory are ignored.

The PREFETCHW instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor and invalidates any other cached copy in anticipation of the line being written to in the future.

The characteristic of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes.

It should be noted that processors are free to speculatively fetch and cache data with exclusive ownership from system memory regions that permit such accesses (that is, the WB memory type). A PREFETCHW instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHW instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHW instruction is also unordered with respect to CLFLUSH instructions, other PREFETCHW instructions, or any other general instruction

It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
FETCH_WITH_EXCLUSIVE_OWNERSHIP (m8);
```

Flags Affected

All flags are affected

C/C++ Compiler Intrinsic Equivalent

```
void _m_prefetchw( void * );
```

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.

RDSEED—Read Random SEED

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C7 /7 RDSEED r16	M	V/V	RDSEED	Read a 16-bit NIST SP800-90B & C compliant random value and store in the destination register.
0F C7 /7 RDSEED r32	M	V/V	RDSEED	Read a 32-bit NIST SP800-90B & C compliant random value and store in the destination register.
REX.W + 0F C7 /7 RDSEED r64	M	V/I	RDSEED	Read a 64-bit NIST SP800-90B & C compliant random value and store in the destination register.

Instruction Operand Encoding

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

Description

Loads a hardware generated random value and store it in the destination register. The random value is generated from an Enhanced NRBG (Non Deterministic Random Bit Generator) that is compliant to NIST SP800-90B and NIST SP800-90C in the XOR construction mode. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random seed value has been returned, otherwise it is expected to loop and retry execution of RDSEED (see Section 1.2).

The RDSEED instruction is available at all privilege levels. The RDSEED instruction executes normally either inside or outside a transaction region.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.B permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF HW_NRND_GEN.ready = 1
    THEN
        CASE of
            osize is 64: DEST[63:0] ← HW_NRND_GEN.data;
            osize is 32: DEST[31:0] ← HW_NRND_GEN.data;
            osize is 16: DEST[15:0] ← HW_NRND_GEN.data;
        ESAC;
        CF ← 1;
    ELSE
        CASE of
            osize is 64: DEST[63:0] ← 0;
            osize is 32: DEST[31:0] ← 0;
            osize is 16: DEST[15:0] ← 0;
        ESAC;
        CF ← 0;
FI;

OF, SF, ZF, AF, PF ← 0;
```

Flags Affected

The CF flag is set according to the result (see the "Operation" section above). The OF, SF, ZF, AF, and PF flags are set to 0.

C/C++ Compiler Intrinsic Equivalent

RDSEED int _rdseed16_step(unsigned short *);

RDSEED int _rdseed32_step(unsigned int *);

RDSEED int _rdseed64_step(unsigned __int64 *);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If the F2H or F3H prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If the F2H or F3H prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
	If the F2H or F3H prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If the F2H or F3H prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If the F2H or F3H prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

CLAC—Clear AC Flag in EFLAGS Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 01 CA CLAC	NP	V/V	SMAP	Clear only the AC flag in the EFLAGS register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

CLAC clears the AC flag bit in EFLAGS/RFLAGS without affecting other bits. Attempt to execute CLAC when CPL > 0 will cause #UD.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
EFLAGS.AC ← 0;
```

Flags Affected

AC cleared; all other flags are unchanged.

C/C++ Compiler Intrinsic Equivalent

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If the CPL > 0.
	If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Virtual-8086 Mode Exceptions

#UD	The CLAC instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If the CPL > 0.
	If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If the CPL > 0.
	If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

STAC—Set AC Flag in EFLAGS Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 01 CB STAC	NP	V/V	SMAP	Set only the AC flag in the EFLAGS register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

STAC sets the AC flag bit in EFLAGS/RFLAGS without affecting other bits. Attempt to execute STAC when CPL > 0 will cause #UD.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

EFLAGS.AC ← 1;

Flags Affected

AC set; all other flags are unchanged.

C/C++ Compiler Intrinsic Equivalent

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If the CPL > 0.
	If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Virtual-8086 Mode Exceptions

#UD	The STAC instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If the CPL > 0.
	If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If the CPL > 0.
	If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

CHAPTER 11
INTEL® PROCESSOR TRACE

11.1 OVERVIEW

Intel® Processor Trace (**Intel PT**) is an extension of Intel® Architecture that captures information about software execution using dedicated hardware facilities that cause only minimal performance perturbation to the software being traced. This information is collected in **data packets**. The first implementation of Intel PT offers **control flow tracing**, which includes in these packets timing and program flow information (e.g. branch targets, branch taken/not taken indications) and program-induced mode related information (e.g. Intel TSX state transitions, CR3 changes). These packets may be buffered internally before being sent to the memory subsystem. After execution completes, debug software can process the trace data and reconstruct the program flow.

11.1.1 Features and Capabilities

Intel PT's control flow trace generates a variety of packets that, when combined with the binaries of a program by a post-processing tool, can be used to produce an exact execution trace. The packets record flow information such as instruction pointers (IP), indirect branch targets, and directions of conditional branches within contiguous code regions (basic blocks).

In addition, the packets record other contextual, timing, and bookkeeping information that enables both functional and performance debugging of applications. Intel PT has several control and filtering capabilities available to customize the tracing information collected and to append other processor state and timing information to enable debugging. For example, there are modes that allow packets to be filtered based on the current privilege level (CPL) or the value of CR3.

Configuration of the packet generation and filtering capabilities are programmed via a set of MSRs. The MSRs generally follow the naming convention of IA32_RTIT_* (Section 11.6 lists the MSRs that support Intel PT).

11.1.1.1 Packet Summary

After a tracing tool has enabled and configured the appropriate MSRs, the processor will collect and generate trace information in the following types of packets (for more details on the packets, see Section 11.4):

- Packet Stream Boundary (**PSB**) packets: PSB packets act as 'heartbeats' that are generated at regular intervals (e.g., every 4K trace packet bytes). These packets allow the packet decoder to find the packet boundaries within the output data stream; a PSB packet should be the first packet that a decoder looks for when beginning to decode a trace.
- Taken Not-Taken (**TNT**) packets: TNT packets track the "direction" of direct conditional branches (taken or not taken).
- Target IP (**TIP**) packets: TIP packets record the target IP of indirect branches, exceptions, interrupts, and other branches or events. These packets can contain the IP, although that IP value may be compressed by eliminating upper bytes that match the last IP. There are various types of TIP packets; they are covered in more detail in Section 11.4.2.2.
- Flow Update Packets (**FUP**): FUPs provide the source IP addresses for asynchronous events (interrupt and exceptions), as well as other cases where the source address cannot be determined from the binary.
- Paging Information Packet (**PIP**): PIPs record modifications made to the CR3 register. This information, along with information from the operating system on the CR3 value of each process, allows the debugger to attribute linear addresses to their correct application source.
- Time-Stamp Counter (**TSC**) packets: TSC packets aid in tracking wall-clock time, and contain some portion of the software-visible time-stamp counter.
- **MODE** packets: These packets provide the decoder with important processor execution information so that it can properly interpret the binary and trace log. MODE packets have a variety of formats that indicate details such as the execution mode (16-bit, 32-bit, or 64-bit).

- Core Bus Ratio (**CBR**) packets: CBR packets contain the core:bus clock ratio.
- Overflow (**OVF**) packets: OVF packets are sent when the processor experiences an internal buffer overflow, resulting in packets being dropped. This packet notifies the decoder of the loss and can help the decoder to respond to this situation.

11.2 INTEL® PROCESSOR TRACE OPERATIONAL MODEL

This section describes the overall Intel Processor Trace mechanism and the essential concepts relevant to how it operates.

11.2.1 Change of Flow Instruction (COFI) Tracing

A basic program block is a section of code where no jumps or branches occur. The instruction pointers (IPs) in this block of code need not be traced, as the processor will execute them from start to end without redirecting code flow. Instructions such as branches, and events such as exceptions or interrupts, can change the program flow. These instructions and events that change program flow are called Change of Flow Instructions (COFI). The program blocks are divided into these three categories:

- Direct transfer COFI.
- Indirect transfer COFI.
- Far transfer COFI.

The following subsections describe the COFI events that result in trace packet generation. Table 11-2 lists branch instruction by COFI types. For detailed description of specific instructions, see *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

Table 11-2. List of Branch Instruction by COFI Type

COFI Type	Instructions
Conditional Branch	JA, JAE, JB, JBE, JC, JCXZ< JECXZ, JRCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ, LOOP, LOOPE, LOOPNE, LOOPNZ, LOOPZ
Unconditional Direct Branch	JMP (E9 xx, EB xx), CALL (E8 xx)
Indirect Branch	JMP (FF /4), CALL (FF /2)
Near Ret	RET (C3, C2 xx)
Far Transfers	INT3, INTn, INTO, IRET, IRETD, IRETQ, JMP (EA xx, REX.W? FF /5), CALL (9A xx, FF /3), RET (CB, CA xx), SYSCALL, SYSRET, SYSENTER, SYSEXIT

11.2.1.1 Direct Transfer COFI

Direct Transfer COFI are relative branches. This means that their target is an IP whose offset from the current IP is embedded in the instruction bytes. It is not necessary to indicate target of these instructions in the trace output since it can be obtained through the source disassembly. Conditional branches need to indicate only whether the branch is taken or not. Unconditional branches do not need any recording in the trace output. There are two sub-categories:

- **Conditional Branch (Jcc, J*CXZ) and LOOP**

  To track this type of instruction, the processor encodes a single bit (taken or not taken — TNT) to indicate the program flow after the instruction. If a branch is taken in an execution flow, the packet encoder will update the target IP internally and emit an encoded bit as "taken" in a TNT packet. If the branch is not taken, the program will simply go to the next instruction, and the TNT packet adds a "not taken" bit.

  Jcc, J*CXZ, and LOOP can be traced with TNT bits. To improve the trace packet output efficiency, the processor will compact several TNT bits into a single packet.

- **Unconditional Direct Jumps**

  There is no trace output for direct unconditional jumps (like JMP near relative or CALL near relative) since they can be directly inferred from the application assembly. Direct unconditional jumps do not generate a TNT bit or a Target IP packet.

11.2.1.2 Indirect Transfer COFI

Indirect transfer instructions involve updating the IP from a register or memory location. Since the register or memory contents can vary at any time during execution, there is no way to know the target of the indirect transfer until the register or memory contents are read. As a result, the disassembled code is not sufficient to determine the target of this type of COFI. Therefore, tracing hardware must send out the destination IP in the trace packet for debug software to determine the target address of the COFI. Note that this IP may be a linear or effective address (see Section 11.3.1.1)

An indirect transfer instruction generates a Target IP Packet (TIP) that contains the target address of the branch. There are two sub-categories:

- **Near JMP Indirect and Near Call Indirect**

  As previously mentioned, the target of an indirect COFI resides in the contents of either a register or memory location. Therefore, the processor must generate a packet that includes this target address to allow the decoder to determine the program flow.

- **Near RET**

  When a CALL instruction executes, it pushes onto the stack the address of the next instruction following the CALL. Upon completion of the call procedure, the RET instruction is often used to pop the return address off of the call stack and redirect code flow back to the instruction following the CALL.

  A RET instruction simply transfers program flow to the address it popped off the stack. Because a called procedure may change the return address on the stack before executing the RET instruction, debug software can be misled if it assumes that code flow will return to the instruction following the last CALL. Therefore, even for near RET, a Target IP Packet may be sent.

  — **RET Compression**

    A special case is applied if the target of the RET is consistent with what would be expected from tracking the CALL stack. If it is assured that the decoder has seen the corresponding CALL (with "corresponding" defined as the CALL with matching stack depth), and the RET target is the instruction after that CALL, the RET target may be "compressed". In this case, only a single TNT bit of "taken" is generated instead of a Target IP Packet. To ensure that the decoder will not be confused in cases of RET compression, only RETs that correspond to CALLs which have been seen since the last PSB packet may be compressed. For details, see "Indirect Transfer Compression for Returns (RET)" in Section 11.4.2.2.

11.2.1.3 Far Transfer COFI

All operations that change the instruction pointer and are not near jumps are "far transfers". This includes exceptions, interrupts, traps, TSX aborts, and instructions that do far transfers.

All far transfers will produce a Target IP (TIP) packet, which provides the destination IP address. For those far transfers that cannot be inferred from the binary source (e.g., asynchronous events such as exceptions and interrupts), the TIP will be preceded by a Flow Update packet (FUP), which provides the source IP address at which the event was taken. Table 11-20 indicates exactly which IP will be included in the FUP generated by a far transfer.

See the packet generation scenarios (Section 11.4.3) for more details on which packets are generated on each variety of far transfer.

11.2.2 Trace Filtering

Intel Processor Trace provides filtering capabilities, by which the debug/profile tool can control what code is traced.

11.2.2.1 Filtering by Current Privilege Level (CPL)

Intel PT provides the ability to specify whether tracing occurs in supervisor (CPL = 0) or user (CPL > 0) modes. It can be configured to generate control flow packets only when CPL = 0; when CPL > 0; or for all values of CPL.

CPL filtering ensures that no IPs or other architectural state information associated with the filtered CPL can be seen in the log. For example, if the processor is configured to trace only when CPL > 0, and software executes SYSCALL (changing the CPL to 0), the destination IP of the SYSCALL will be suppressed from the generated packet (see the discussion of TIP.PGD in Section 11.4.2.5).

It should be noted that CPL is always 0 in real-address mode and that CPL is always 3 in virtual-8086 mode. To trace code in these modes, filtering should be configured accordingly.

When software is executing in a non-enabled CPL, ContextEn is cleared. See Section 11.2.3.1 for details.

11.2.2.2 Filtering by CR3

Intel PT supports a CR3-filtering mechanism by which control-flow packet generation can be enabled or disabled based on the value of CR3. A debugger can use CR3 filtering to trace only a single application without context switching the state of the RTIT MSRs. To the reconstruction of traces from software with multiple threads, debug software may wish to context-switch the state of the RTIT MSRs (if the operating system does not provide context-switch support) to separate the output for the different threads (see Section 11.3.4, "Context Switch Consideration").

To trace for only a single CR3 value, software can write that value to the IA32_RTIT_CR3_MATCH MSR, while setting IA32_RTIT_CTL.CR3Filter. When CR3 value does not match IA32_RTIT_CR3_MATCH and IA32_RTIT_CTL.CR3Filter is 1, ContextEn is forced to 0, and control-flow packets will not be generated. (Some other packets can be generated when ContextEn is 0; see Section 11.2.3.3 for details.) When CR3 does match IA32_RTIT_CR3_MATCH (or when IA32_RTIT_CTL.CR3Filter is 0), CR3 filtering does not force ContextEn to 0 (although it could be 0 due to other filters or modes).

CR3 matches IA32_RTIT_CR3_MATCH if the two registers are identical for bits 63:5; the lower 5 bits of CR3 and IA32_RTIT_CR3_MATCH are ignored. CR3 filtering is independent of the value of CR0.PG.

When CR3 filtering is in use, PIP packets may still be seen in the log if the processor is configured to trace when CPL = 0 (IA32_RTIT_CTL.OS = 1). If not, no PIP packets will be seen.

11.2.3 Packet Generation Enable Controls

Intel Processor Trace includes a variety of controls that determine whether a packet is generated. In general, most packets are sent only if Packet Enable (**PacketEn**) is set. PacketEn is an internal state maintained in hardware in response to software configurable enable controls, PacketEn is not visible to software directly. The relationship of PacketEn to the software-visible controls in the configuration MSRs is described in this section.

11.2.3.1 Packet Enable (PacketEn)

When PacketEn is set, the processor is in the mode that Intel PT is monitoring and all packets can be generated to log what is being executed. PacketEn is composed of other states according to this relationship:

```
PacketEn = TriggerEn AND ContextEn
```

These constituent controls are detailed in the following subsections.

PacketEn ultimately determines when the processor is tracing. When PacketEn is set, all control flow packets are enabled. When PacketEn is clear, no control flow packets are generated, though other packets (timing and bookkeeping packets) may still be sent. See Section 11.2.4 for details of PacketEn and packet generation.

11.2.3.2 Trigger Enable (TriggerEn)

Trigger Enable (**TriggerEn**) is the primary indicator that trace packet generation is active. TriggerEn is set when IA32_RTIT_CTL.TraceEn is set, and neither IA32_RTIT_STATUS.Stopped nor IA32_RTIT_STATUS.Error is set.

The processor may not update ContextEn when TriggerEn=0. The processor guarantees that ContextEn is correctly evaluated only when TriggerEn = 1.

Software can discover the current TriggerEn value by reading the IA32_RTIT_STATUS.TriggerEn bit. When TriggerEn is clear, tracing is inactive and no packets are generated.

11.2.3.3 Context Enable (ContextEn)

Context Enable (**ContextEn**) indicates whether the processor is in the state or mode that software configured hardware to trace. For example, if execution with CPL = 0 code is not being traced (IA32_RTIT_CTL.OS = 0), then ContextEn will be 0 when the processor is in CPL0.

Software can discover the current ContextEn value by reading the IA32_RTIT_STATUS.ContextEn bit. ContextEn is defined as follows:

```
ContextEn = !((IA32_RTIT_CTL.OS = 0 AND CPL = 0) OR
(IA32_RTIT_CTL.USER = 0 AND CPL > 0) OR
(IA32_RTIT_CTL.CR3Filter = 1 AND IA32_RTIT_CR3_MATCH does not match CR3)
```

If the clearing of ContextEn causes PacketEn to be cleared, a Packet Generation Disable (TIP.PGD) packet is generated, but its IP payload is suppressed. If the setting of ContextEn causes PacketEn to be set, a Packet Generation Enable (TIP.PGE) packet is generated.

When ContextEn is 0, control flow packets TNT, FUP, TIP, PIP, MODE) are not generated, and no LIPs are exposed. For details of which packets are generated only when ContextEn is set, see Section 11.4.1.

The processor does not update ContextEn when TriggerEn = 0.

11.2.4 Packet Output to Memory

Trace output is written to memory in a collection of variable-sized regions of physical memory. These regions are linked together by tables of pointers to those regions, referred to as Table of Physical Addresses (**ToPA**). The trace output stores bypass the caches and the TLBs, but are not serializing. This is intended to minimize the performance impact of the output.

11.2.4.1 Table of Physical Addresses (ToPA)

The ToPA mechanism uses a linked list of tables; see Figure 11-1 for an illustrative example. Each entry in the table contains some attribute bits, a pointer to an output region, and the size of the region. The last entry in the table may hold a pointer to the next table. This pointer can either point to the top of the current table (for circular array) or to the base of another table. The table size is not fixed, since the link to the next table can exist at any entry.

The processor treats the various output regions referenced by the ToPA table(s) as a unified buffer. This means that a single packet may span the boundary between one output region and the next.

The ToPA mechanism is controlled by three values maintained by the processor:

- **proc_trace_table_base.**
  This is the physical address of the base of the current ToPA table. When tracing is enabled, the processor loads this value from the IA32_RTIT_OUTPUT_BASE MSR. While tracing is enabled, the processor updates the IA32_RTIT_OUTPUT_BASE MSR with changes to proc_trace_table_base, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc_trace_table_base.
- **proc_trace_table_offset.**
  This indicates the entry of the current table that is currently in use. (This entry contains the address of the current output region.) When tracing is enabled, the processor loads this value from bits 31:7 (MaskOrTableOffset) of the IA32_RTIT_OUTPUT_MASK_PTRS. While tracing is enabled, the processor updates IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTableOffset with changes to proc_trace_table_offset, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc_trace_table_offset.
- **proc_trace_output_offset.**
  This a pointer into the current output region and indicates the location of the next write. When tracing is enabled, the processor loads this value from bits 63:32 (OutputOffset) of the IA32_RTIT_OUTPUT_MASK_PTRS. While tracing is enabled, the processor updates IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset with changes to proc_trace_output_offset, but these updates

may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc_trace_output_offset.

Figure 11-1 provides an illustration (not to scale) of the table and associated pointers.

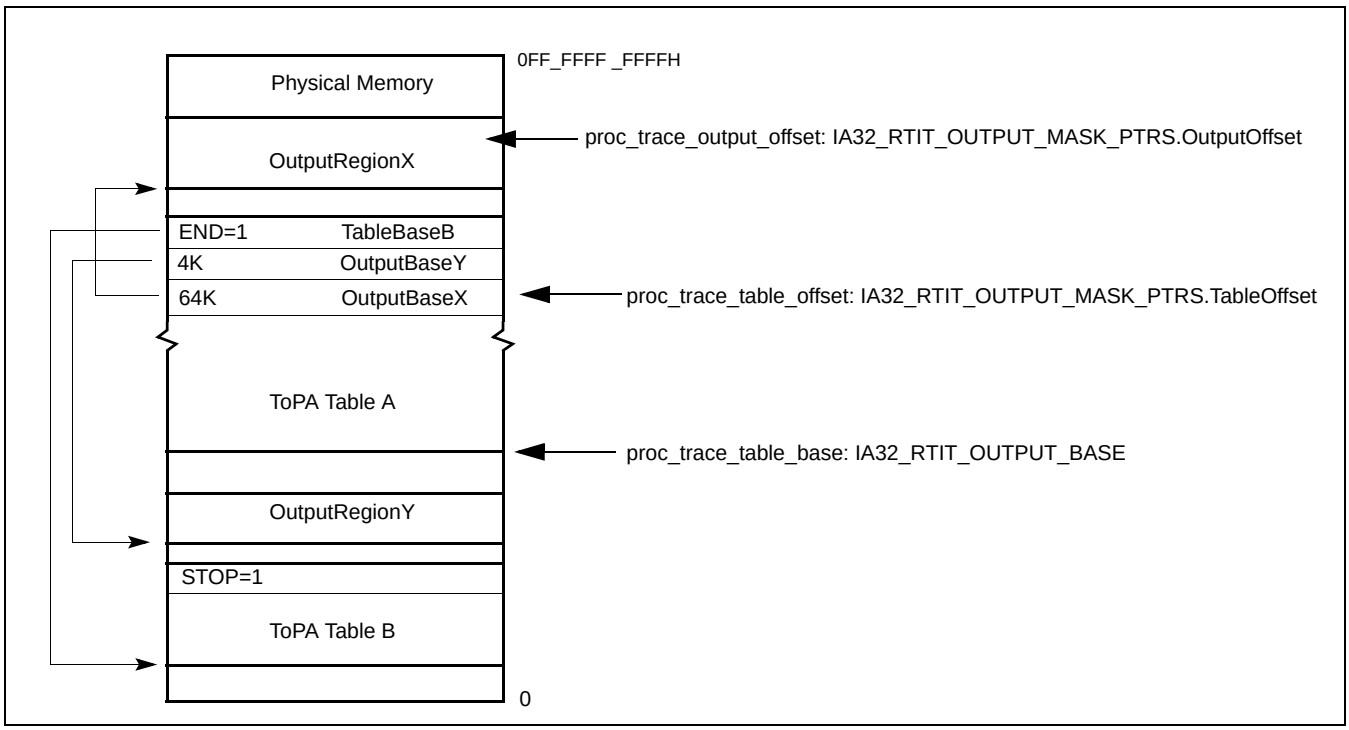


Figure 11-1. ToPA Memory Illustration

With the ToPA mechanism, the processor writes packets to the current output region (identified by proc_trace_table_base and the proc_trace_table_offset). The offset within that region to which the next byte will be written is identified by proc_trace_output_offset. When that region is filled with packet output (thus proc_trace_output_offset = RegionSize–1), proc_trace_table_offset is moved to the next ToPA entry, proc_trace_output_offset is set to 0, and packet writes begin filling the new output region specified by proc_trace_table_offset.

Eventually, the regions represented by all entries in the table may become full, and the final entry of the table is reached. An entry can be identified as the final entry because it has either the END or STOP attribute. The END attribute indicates that the address in the entry does not point to another output region, but rather to another ToPA table. The STOP attribute indicates that tracing will be disabled once the corresponding region is filled. See Section 11.2.4.1 for details on STOP.

When an END entry is reached, the processor loads proc_trace_table_base with the base address held in this END entry, thereby moving the current table pointer to this new table. The proc_trace_table_offset is reset to 0, as is the proc_trace_output_offset, and packet writes will resume at the base address indicated in the first entry.

If the table has no STOP or END entry, and trace-packet generation remains enabled, eventually the maximum table size will be reached (proc_trace_table_offset = FFFFFFFFH). In this case, the proc_trace_table_offset and proc_trace_output_offset are reset to 0 (wrapping back to the beginning of the current table) once the last output region is filled.

It is important to note that processor updates to the IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs are asynchronous to instruction execution. Thus, reads of these MSRs while Intel PT is enabled may return stale values. Like all IA32_RTIT_* MSRs, the values of these MSRs should not be trusted or saved unless trace packet generation is first disabled by clearing IA32_RTIT_CTL.TraceEn. This ensures that all internally buffered packet data are written to memory. When TraceEn is 0, the values of the IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSR are up to date and do not change. A store

fence or serializing instruction following the clearing of TraceEn may be required to ensure that trace output data are globally observed.[1]

The processor may cache internally any number of entries from the current table or from tables that it references (directly or indirectly). If tracing is enabled, the processor may ignore or delay detection of modifications to these tables. To ensure that table changes are detected by the processor in a predictable manner, software should clear TraceEn before modifying the current table (or tables that it references) and only then re-enable packet generation.

As packets are written out to memory, each store derives its physical address as follows:

```
trace_store_phys_addr = Base address from current ToPA table entry +
proc_trace_output_offset
```

There is no guarantee that a packet will be written to memory after some fixed number of cycles after a packet-producing instruction executes. The only way to assure that all packets generated can be seen in memory is to clear TraceEn; doing so ensures that all buffered packets are written to memory.

Single Output Region ToPA Implementation

The first processor generation to implement Intel PT supports only ToPA configurations with a single ToPA entry followed by an END entry that points back to the first entry (creating one circular output buffer). Such processors enumerate CPUID.(EAX=14H,ECX=0):EBX[bit 1] as 0.

ToPA Table Entry Format

The format of ToPA table entries is shown in Figure 11-2. The size of the address field is determined by the processor's physical-address width (MAXPHYADDR) in bits, as reported in CPUID.80000008H:EAX[7:0].

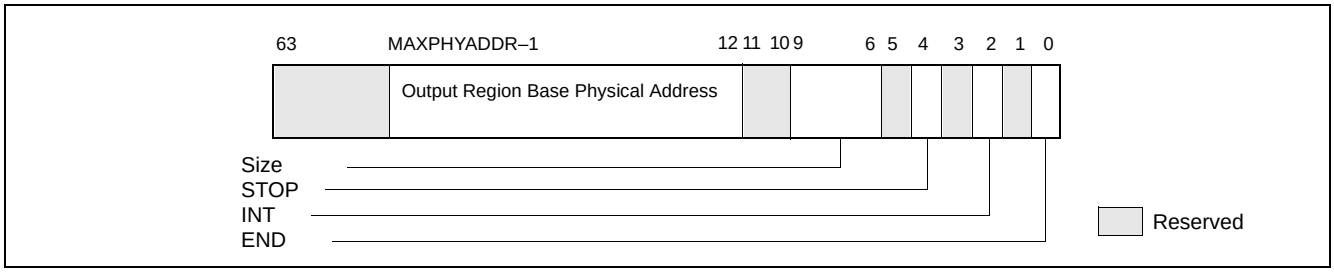


Figure 11-2. Layout of ToPA Table Entry

Table 11-3 describes the details of the ToPA table entry fields. If reserved bits are set to 1, an error is signaled.

Table 11-3. ToPA Table Entry Fields

ToPA Entry Field	Description
Output Region Base Physical Address	If END=0, this is the 4K-aligned base physical address of the output region specified by this entry. Note that all regions must be aligned based on their size, not just 4K. Thus a 2M region must have bits 20:12 clear. If the region is not properly aligned, an operational error will be signaled when the entry is reached. If END=1, this is the 4K-aligned base physical address of the next ToPA table (which may be the base of the current table, or the first table in the linked list if a circular buffer is desired). If the processor supports only a single ToPA output region (see above), this address must be the value currently in the IA32_RTIT_OUTPUT_BASE MSR.
Size	Indicates the size of the associated output region. Encodings are: 0: 4K, 1: 8K, 2: 16K, 3: 32K, 4: 64K, 5: 128K, 6: 256K, 7: 512K, 8: 1M, 9: 2M, 10: 4M, 11: 8M, 12: 16M, 13: 32M, 14: 64M, 15: 128M This field is ignored if END=1.

1. Although WRMSR is a serializing instruction, the execution of WRMSR that forces packet writes by clearing TraceEn does not itself cause these writes to be globally observed.

Table 11-3. ToPA Table Entry Fields

ToPA Entry Field	Description
STOP	When the output region indicated by this entry is filled, software should disable packet generation. This will be accomplished by setting IA32_RTIT_STATUS.Stopped, which clears TriggerEn. This bit must be 0 if END=1; otherwise it is treated as reserved bit violation (see ToPA Errors)
INT	When the output region indicated by this entry is filled, signal Perfmon LVT interrupt. Note that if both INT and STOP are set in the same entry, the STOP will happen before the INT. Thus the interrupt handler should expect that the IA32_RTIT_STATUS.Stopped bit will be set, and will need to be reset before tracing can be resumed. This bit must be 0 if END=1; otherwise it is treated as reserved bit violation (see ToPA Errors)
END	If set, indicates that this is an END entry, and thus the address field points to a table base rather than an output region base. If END=1, INT and STOP must be set to 0; otherwise it is treated as reserved bit violation (see ToPA Errors). The Size field is ignored in this case. If the processor supports only a single ToPA output region (see above), END must be 1 in any ToPA entry other than the first (whenever proc_trace_table_offset differs from the value in the IA32_RTIT_OUTPUT_BASE MSR.

ToPA STOP

Each ToPA entry has a STOP bit. If this bit is set, the processor will set the IA32_RTIT_STATUS.Stopped bit when the corresponding trace output region is filled. This will clear TriggerEn and thereby cease packet generation. See Section 11.2.5.3 for details on IA32_RTIT_STATUS.Stopped. This sequence is known as "ToPA Stop"

No TIP.PGD packet will be seen in the output when the ToPA stop occurs, since the disable happens only when the region is already full. When this occurs, any packets remaining in internal buffers are lost and cannot be recovered.

When ToPA stop occurs, the IA32_RTIT_OUTPUT_BASE MSR will hold the base address of the table whose entry had STOP=1. IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTableOffset will hold the index value for that entry, and the IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset should be set to the size of the region.

ToPA PMI

Each ToPA entry has an INT bit. If this bit is set, the processor will signal a performance-monitoring interrupt (PMI) when the corresponding trace output region is filled. This interrupt is not precise, and it is thus likely that writes to the next region will occur by the time the interrupt is taken.

A usage model envisioned for this attribute is for software to copy output data to external memory before the output region is full.

The following steps should be taken to configure this interrupt:

1. Enable PMI via the LVT Performance Monitor register (at MMIO offset 340H in xAPIC mode; via MSR 384H in x2APIC mode). See *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B* for more details on this register. For ToPA PMI, set all fields to 0, save for the interrupt vector, which can be selected by software.
2. Set the interrupt flag by executing STI.
3. Set the INT bit in the ToPA entry of interest and enable packet generation, using the ToPA output option. Thus, TraceEn=ToPA=1 in the IA32_RTIT_CTL MSR.

Once the INT region has been filled with packet output data, the interrupt will be signaled. This PMI can be distinguished from others by checking bit 55 (Trace_ToPA_PMI) of the IA32_PERF_GLOBAL_STATUS MSR (MSR 38EH). Once the ToPA PMI has been serviced, this bit can be cleared by writing bit 55 of the IA32_PERF_GLOBAL_OVF_CTL MSR.

Note that no "freezing" takes place with the ToPA PMI. Thus, packet generation is not frozen, and the interrupt handler will be traced (though filtering can prevent this). Further, the setting of IA32_DEBUGCTL.Freeze_Perfmon_on_PMI is ignored and performance counters are not frozen by a ToPA PMI.

Assuming the PMI handler wishes to read any buffered packets for persistent output, software should first disable packet generation by clearing TraceEn. This ensures that all buffered packets are written to memory and avoids tracing of the PMI handler. The configuration MSRs can then be used to determine where tracing has stopped. If

packet generation is disabled by the handler, it should then be manually re-enabled before the IRET if continued tracing is desired.

ToPA PMI and Single Output Region ToPA Implementation

A processor that supports only a single ToPA output region implementation (such that only one output region is supported; see above) will attempt to signal a ToPA PMI interrupt before the output wraps and overwrites the top of the buffer. To support this functionality, the PMI handler should disable packet generation as soon as possible.

Due to PMI skid, it is possible, in rare cases, that the wrap will have occurred before the PMI is delivered. Software can avoid this by setting the STOP bit in the ToPA entry (see Table 11-3); this will disable tracing once the region is filled, and no wrap will occur. This approach has the downside of disabling packet generation so that some of the instructions that led up to the PMI will not be traced. If the PMI skid is significant enough to cause the region to fill and tracing to be disabled, the PMI handler will need to clear the IA32_RTIT_STATUS.Stopped indication before tracing can resume.

ToPA Errors

When a malformed ToPA entry is found, an **operation error** results (see Section 11.3.7). A malformed entry can be any of the following:

1. **ToPA entry reserved bit violation**.
   This describes cases where a bit marked as reserved in Section 11.2.4.1 above is set to 1.
2. **ToPA alignment violation**.
   This includes cases where illegal ToPA entry base address bits are set to 1:
   a. ToPA table base address is not 4KB-aligned. The table base can be from a WRMSR to IA32_RTIT_OUTPUT_BASE, or from a ToPA entry with END=1.
   b. ToPA entry base address is not aligned to the ToPA entry size (e.g., a 2MB region with base address[20:12] not equal to 0).
   c. ToPA entry base address sets upper physical address bits not supported by the processor.
3. **Illegal ToPA Output Offset** (if IA32_RTIT_STATUS.Stopped=0).
   IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset is greater than or equal to the size of the current ToPA output region size.
4. **ToPA rules violations**.
   These are similar to ToPA entry reserved bit violations; they are cases when a ToPA entry is encountered with illegal field combinations. They include the following:
   a. Setting the STOP or INT bit on an entry with END=1.
   b. Setting the END bit in entry 0 of a ToPA table.
   c. On processors that support only a single ToPA entry (see above), two additional illegal settings apply:
      i) ToPA table entry 1 with END=0.
      ii) ToPA table entry 1 with base address not matching the table base.

In all cases, the error will be logged by setting IA32_RTIT_STATUS.Error, thereby disabling tracing when the problematic ToPA entry is reached (when proc_trace_table_offset points to the entry containing the error). Any packet bytes that are internally buffered when the error is detected may be lost.

Note that operational errors may also be signaled due to attempts to access restricted memory. See Section 11.2.4.2 for details.

A tracing software have a range of flexibility using ToPA to manage the interaction of Intel PT with application buffers, see Section 11.5.

11.2.4.2 Restricted Memory Access

Packet output cannot be directed to any regions of memory that are restricted by the platform. In particular, all memory accesses on behalf of packet output are checked against the SMRR regions. If there is any overlap with

these regions, trace data collection will not function properly. Exact processor behavior is implementation-dependent; Table 11-4 summarizes several scenarios.

Table 11-4. Behavior on Restricted Memory Access

Scenario	Description
ToPA output region overlaps with SMRR	Stores to the restricted memory region will be dropped, and that packet data will be lost. Any attempt to read from that restricted region will return all 1s. The processor also may signal an error (Section 11.3.7) and disable tracing when the output pointer reaches the restricted region. If packet generation remains enabled, then packet output may continue once stores are no longer directed to restricted memory (on wrap, or if the output region is larger than the restricted memory region).
ToPA table overlaps with SMRR	The processor will signal an error (Section 11.3.7) and disable tracing when the ToPA read pointer (IA32_RTIT_OUTPUT_BASE + (proc_trace_table_offset « 3)) enters the restricted region.

It should also be noted that packet output should not be routed to the 4KB APIC MMIO region, as defined by the IA32_APIC_BASE MSR. For details about the APIC, refer to *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. No error is signaled for this case.

Modifications to Restricted Memory Regions

It is recommended that software disable packet generation before modifying the SMRRs to change the scope of the SMRR regions. This is because the processor reserves the right to cache any number of ToPA table entries internally, after checking them against restricted memory ranges. Once cached, the entries will not be checked again, meaning one could potentially route packet output to a newly restricted region. Software can ensure that any cached entries are written to memory by clearing IA32_RTIT_CTL.TraceEn.

11.2.5 Enabling and Configuration MSRs

11.2.5.1 General Considerations

Trace packet generation is enabled and configured by a collection of model-specific registers (MSRs), which are detailed below. Some notes on the configuration MSR behavior:

- If Intel Processor Trace is not supported by the processor (see Section 11.3.1), RDMSR or WRMSR of the IA32_RTIT_* MSRs will cause #GP.
- A WRMSR to any of these configuration MSRs that begins and ends with IA32_RTIT_CTL.TraceEn set will #GP fault. Packet generation must be disabled before the configuration MSRs can be changed.

  Note: Software may write the same value back to IA32_RTIT_CTL without #GP, even if TraceEn=1.
- All configuration MSRs for Intel PT are duplicated per logical processor
- For each configuration MSR, any MSR write that attempts to change bits marked reserved, or utilize encodings marked reserved, will cause a #GP fault.

11.2.5.2 IA32_RTIT_CTL MSR

IA32_RTIT_CTL, at address 570H, is the primary enable and control MSR for trace packet generation. Bit positions are listed in Table 11-5.

Table 11-5. IA32_RTIT_CTL MSR

Position	Bit Name	At Reset	Bit Description
0	TraceEn	0	If 1, enables tracing; else tracing is disabled if 0 When this bit transitions from 1 to 0, all buffered packets are written to their output destination. When this bit transitions from 0 to 1, a series of packets may be generated. This may include PSB, along with associated status packets (see Section 11.4.2.3), or may include only a TSC and CBR packet (see Section 11.4.2.9 and Section 11.4.2.10). If changing this bit changes PacketEN, a TIP.PGE or TIP.PGD will be generated. In the TIP.PGE case, a MODE packet will precede it, see Section 11.4.2.8.
1	Reserved	0	Must be 0
2	OS	0	0: Packet generation is disabled when CPL = 0 1: Packet generation may be enabled when CPL = 0
3	User	0	0: Packet generation is disabled when CPL > 0 1: Packet generation may be enabled when CPL > 0
6:4	Reserved	0	Must be 0
7	CR3Filter	0	0: Disables CR3 filtering 1: Enables CR3 filtering
8	ToPA	0	1: ToPA output scheme enabled (see Section 11.2.4.1) WRMSR to IA32_RTIT_CTL that sets TraceEn but clears this bit causes #GP.
9	Reserved	0	Must be 0
10	TSCEn	0	0: Disable TSC packets 1: Enable TSC packets (see Section 11.4.2.10)
11	DisRETC	0	0: Enable RET compression 1: Disable RET compression (see Section 11.2.1.2)
12	Reserved	0	Must be 0
13	Reserved	0	WRMSR to IA32_RTIT_CTL that sets TraceEn but clears this bit causes #GP.
63:14	Reserved	0	Must be 0

Enabling Packet Generation

When TraceEn transitions from 0 to 1, packet generation is enabled, and a series of packets may be generated. These packets help ensure that the decoder is aware of the state of the processor when the trace begins, and that it can keep track of any timing or state changes that may have occurred while packet generation was disabled. This may be a full PSB+ (see Section 11.4.2.12), or it may be a TSC (see Section 11.4.2.10) followed by CBR (see Section 11.4.2.9), if those packets are enabled.

In addition to the packets above, once PacketEn (Section 11.2.3.1) transitions from 0 to 1 (which may happen immediately, depending on filtering settings), a MODE.Exec packet (Section 11.4.2.8) followed by a TIP.PGE packet (Section 11.4.2.3) will be generated. The TIP.PGE and MODE packets could come before or after the PSB packet, the TSC packet, or the CBR packet.

When TraceEn is set, the processor may read ToPA entries from memory and cache them internally. For this reason, software should disable packet generation before making modifications to the ToPA tables (or changing the configuration of restricted memory regions). See Section 11.4.3 for more details of packets that may be generated with modifications to TraceEn.

Disabling Packet Generation

A WRMSR that clears TraceEn causes any buffered packets to be written to memory. After software disables packet generation by clearing TraceEn, all packets have been written to memory and that the output MSRs (IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS) have stable values. (As noted earlier a store

fence or serializing instruction may be required to ensure that trace output data are globally observed.[1]) No special packets are generated by disabling packet generation, though a TIP.PGD may result if PacketEn=1 at the time of disable.

Other Writes to IA32_RTIT_CTL

Any attempt to modify IA32_RTIT_CTL while TraceEn is set will result in a general-protection fault (#GP) unless the same write also clears TraceEn. However, writes to IA32_RTIT_CTL that do not modify any bits will not cause a #GP, even if TraceEn remains set.

11.2.5.3 IA32_RTIT_STATUS MSR

The IA32_RTIT_STATUS MSR is readable and writable by software, but some bits (ContextEn, TriggerEn) are read-only and cannot be directly modified. The WRMSR instruction ignores these bits in the source operand (attempts to modify these bits are ignored and do not cause WRMSR to fault).

This MSR can only be written when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP). The processor does not modify the value of this MSR while TraceEn is 0 (software can modify it with WRMSR).

Table 11-6. IA32_RTIT_STATUS MSR

Position	Bit Name	At Reset	Bit Description
0	Reserved	0	Writes are ignored.
1	ContextEn	0	The processor sets this bit to indicate that tracing is allowed for the current context. See Section 11.2.3.3. Writes are ignored.
2	TriggerEn	0	The processor sets this bit to indicate that tracing is enabled. See Section 11.2.3.2. Writes are ignored.
3	Reserved	0	Must be 0.
4	Error	0	The processor sets this bit to indicate that an operational error has been encountered. When this bit is set, TriggerEn is cleared to 0 and packet generation is disabled. For details, see "ToPA Errors" in Section 11.2.4.1. When TraceEn is cleared, software can write this bit. Once it is set, only software can clear it. It is not recommended that software ever set this bit, except in cases where it is restoring a prior saved state
5	Stopped	0	The processor sets this bit to indicate that a ToPA Stop condition has been encountered. When this bit is set, TriggerEn is cleared to 0 and packet generation is disabled. For details, see "ToPA STOP" in Section 11.2.4.1. When TraceEn is cleared, software can write this bit. Once it is set, only software can clear it. It is not recommended that software ever set this bit, except in cases where it is restoring a prior saved state.
63:6	Reserved	0	Must be 0.

11.2.5.4 IA32_RTIT_CR3_MATCH MSR

When IA32_RTIT_CTL.CR3Filter is 1, ContextEn is set on only if CR3 matches the IA32_RTIT_CR3_MATCH MSR. CR3 matches IA32_RTIT_CR3_MATCH if the two registers are identical for bits 63:5; the lower 5 bits of CR3 and IA32_RTIT_CR3_MATCH are not compared. For more details, see Section 11.2.2.2.

This MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP). IA32_RTIT_CR3_MATCH[4:0] are reserved and must be 0; an attempt to set those bits using WRMSR causes a #GP.

1. Although WRMSR is a serializing instruction, the execution of WRMSR that forces packet writes by clearing TraceEn does not itself cause them to be globally observed.

11.2.5.5 IA32_RTIT_OUTPUT_BASE MSR

This MSR is used to configure the output region of internally-buffered packets. The size of the address field is determined by the maximum physical address width (MAXPHYADDR), as reported by CPUID.80000008H:EAX[7:0].

The processor updates this MSR while when packet generation is enabled, and those updates are asynchronous to instruction execution. Therefore, the values in this MSR should be considered unreliable unless packet generation is disabled (IA32_RTIT_CTL.TraceEn = 0).

This MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP).

Table 11-7. IA32_RTIT_OUTPUT_BASE MSR

Position	Bit Name	At Reset	Bit Description
6:0	Reserved	0	Must be 0.
MAXPHYADDR-1:7	BasePhysAddr	0	The base physical address of the current ToPA table. The address must be 4K aligned. Writing an address in which bits 11:7 are non-zero will not cause a #GP, but an operational error will be signaled once TraceEn is set. See "ToPA Errors" in Section 11.2.4.1 as well as Section 11.3.7.
63:MAXPHYADDR	Reserved	0	Must be 0.

11.2.5.6 IA32_RTIT_OUTPUT_MASK_PTRS MSR

This MSR holds the pointers that indicate the ToPA entry that is currently in use and the offset into that entry's output region to which packets are being written. See Section 11.2.4.1 for details.

The processor updates this MSR while when packet generation is enabled, and those updates are asynchronous to instruction execution. Therefore, the values in this MSR should be considered unreliable unless packet generation is disabled (IA32_RTIT_CTL.TraceEn = 0).

This MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP).

Table 11-8. IA32_RTIT_OUTPUT_MASK_PTRS MSR

Position	Bit Name	At Reset	Bit Description
6:0	LowerMask	7FH	Forced to 1, writes are ignored.
31:7	MaskOrTableOffset	0	This field holds bits 27:3 of the offset pointer into the current ToPA table. This value can be added to the IA32_RTIT_OUTPUT_BASE value to produce a pointer to the current ToPA table entry, which itself is a pointer to the current output region. In this scenario, the lower 7 reserved bits are ignored. This field supports tables up to 256 MBytes in size.
63:32	OutputOffset	0	This field holds bits 31:0 of the offset pointer into the current ToPA output region. This value will be added to the output region base field, found in the current ToPA table entry, to form the physical address at which the next byte of trace output data will be written. This value must be less than the ToPA entry size, otherwise an operational error (Section 11.3.7) will be signaled when TraceEn is set.

11.2.6 Interaction of Intel® Processor Trace and Other Processor Features

11.2.6.1 Intel® Transactional Synchronization Extensions (Intel® TSX)

The operation of Intel TSX is described in Chapter 14 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*. For tracing purpose, packet generation does not distinguish between hardware lock elision (HLE) and restricted transactional memory (RTM), but speculative execution does have impacts on the trace output. Specifically, packets are generated as instructions complete, even for instructions in a transactional region that is later aborted. For this reason, debugging software will need indication of the beginning and end of a transactional region; this will allow software to understand when instructions are part of a transactional region and whether that region has been committed.

To enable this, TSX information is included in a MODE packet leaf. The mode bits in the leaf are:

- **InTX**: Set to 1 on an TSX transaction begin, and cleared on transaction commit or abort.
- **TXAbort**: Set to 1 only when InTX transitions from 1 to 0 on an abort. Cleared otherwise.

This MODE packet will be sent each time the transaction status changes. See Table 11-9 for details.

Table 11-9. TSX Packet Scenarios

TSX Event	Instruction	Packets
Transaction Begin	Either XBEGIN or XACQUIRE lock (the latter if executed transactionally)	MODE(TXAbort=0, InTX=1), FUP(CurrentIP).
Transaction Commit	Either XEND or XRELEASE lock, if transactional execution ends. This happens only on the outermost commit	MODE(TXAbort=0, InTX=0), FUP(CurrentIP)
Transaction Abort	XABORT or other transactional abort	MODE(TXAbort=1, InTX=0), FUP(CurrentIP), TIP(TargetIP)
Other	One of the following: ▪ Nested XBEGIN or XACQUIRE lock ▪ An outer XACQUIRE lock that doesn't begin a transaction (InTX not set) ▪ Non-outermost XEND or XRELEASE lock	None. No change to TSX mode bits for these cases

The CurrentIP listed above is the IP of the associated instruction. The TargetIP is the IP of the next instruction to be executed; for HLE, this is the XACQUIRE lock; for RTM, this is the fallback handler.

Intel PT stores are non-transactional, and thus packet writes are not rolled back on TSX abort.

11.2.6.2 System Management Mode (SMM)

SMM code has special privileges that non-SMM code does not have. Intel Processor Trace can be used to trace SMM code, but special care is taken to ensure that SMM handler context is not exposed in any non-SMM trace collection and that packet output from non-SMM code cannot be written into memory space protected by SMRR.

SMM is entered via a system management interrupt (SMI). SMI delivery saves the value of IA32_RTIT_CTL.TraceEn into SMRAM and then clears it, thereby disabling packet generation.

The saving and clearing of IA32_RTIT_CTL.TraceEn ensures two things:

1. All internally buffered packet data is written to memory before entering SMM (see Section 11.2.5.2).
2. Packet generation ceases before entering SMM, so any tracing that was configured outside SMM does not continue into SMM. No SMM instruction pointers or other state will be exposed in the non-SMM trace.

When the RSM instruction is executed to return from SMM, the TraceEn value that was saved by SMI delivery is restored, allowing tracing to be resumed. As is done any time packet generation is enabled, ContextEn is re-evaluated, based on the values of CPL, CR3, etc., established by RSM.

Like other interrupts, delivery of an SMI produces a FUP containing the IP of the next instruction to execute. By toggling TraceEn, SMI and RSM can produce TIP.PGD and TIP.PGE packets, indicating that tracing was disabled or re-enabled. Table 11-10 shows an example of the packets that can be expected when an SMI occurs while the processor is tracing (PacketEn = 1) software outside SMM.

Table 11-10. SMI/RSM Packets When Trace Packet Generation is Enabled Outside SMM

Code Flow	Packets
... Non-SMM Code 1004H ADD %ebx, %eax ; #SMI arrives	... Non-SMM Packets FUP(1006H), TIP.PGD()
38000H JMP bar ; Enters SMM handler ... SMM code 38500H RSM	 TIP.PGE(1006H)

Table 11-10. SMI/RSM Packets When Trace Packet Generation is Enabled Outside SMM

Code Flow	Packets
1006H SUB %ebx, %ebp ... More Non-SMM Code	... More non-SMM packets

Note that packets generated by RSM are undefined, and it is recommended that TraceEn e cleared before executing RSM. Further, on processors that restrict use of Intel PT with LBRs (see Section 11.3.1.2), any RSM that results in enabling of both will cause a shutdown.

11.2.6.3 Virtual-Machine Extensions (VMX)

Initial implementations of Intel Processor Trace do not support tracing in VMX operation. Execution of the VMXON instruction clears TraceEn. An attempt to set IA32_RTIT_CTL.TraceEn using WRMSR in VMX operation causes a general-protection fault (#GP).

This implies that these implementations do not support Intel PT is not supported in a virtualized environment. Future implementations may relax this restriction.

11.2.6.4 SENTER/ENTERACCS and ACM

GETSEC[SENTER] and GETSEC[ENTERACCS] instructions clear TraceEn, and it is not restored when those instruction complete. SENTER also causes TraceEn to be cleared on other logical processors when they rendezvous and enter the SENTER sleep state. In these two cases, the disabling of packet generation is not guaranteed to write buffered packets to memory. Some packets may be dropped.

When executing an authenticated code module (ACM), packet generation is silently disabled during ACRAM setup. TraceEn will be cleared, but no TIP.PGD packet is generated. After completion of the module, the TraceEn value will be restored. There will be no TIP.PGE packet, but timing packets, like TSC and CBR, may be produced.

11.3 CONFIGURATION AND PROGRAMMING GUIDELINE

11.3.1 Detection of Intel Processor Trace and Capability Enumeration

Processor support for Intel Processor Trace is indicated by CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1. CPUID function 14H is dedicated to enumerate the resource and capability of processors that report CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1. Different processor generations may have architecturally-defined variation in capabilities. Table 11-11 describes details of the enumerable capabilities that software must use across generations of processors that support Intel Processor Trace.

Table 11-11. CPUID Leaf 14H Enumeration of Intel Processor Trace Capabilities

CPUID.(EAX=14H,ECX=0)		Name	Description Behavior
Register	Bits		
EAX	31:0	Maximum valid sub-leaf Index	Specifies the index of the maximum valid sub-leaf for this CPUID leaf
EBX	0	CR3 Filtering Support	1: Indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed. See Section 11.2.5. 0: Indicates that writes that set IA32_RTIT_CTL.CR3Filter to 1, or any access to IA32_RTIT_CR3_MATCH, will #GP fault.
	31:1	Reserved	

Table 11-11. CPUID Leaf 14H Enumeration of Intel Processor Trace Capabilities

CPUID.(EAX=14H,ECX=0)		Name	Description Behavior
Register	Bits		
ECX	0	ToPA Output Supported	1: Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme (Section 11.2.4.1) IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed. 0: Enabling tracing (TraceEn=1) with IA32_RTIT_CTL.ToPA=1 or IA32_RTIT_OUTPUT_MASK_PTRS. MSRs will #GP fault.
	1	ToPA Tables Allow Multiple Output Entries	1: ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. 0: ToPA tables can hold only one output entry, which must be followed by an END=1 entry which points back to the base of the table. Further, ToPA PMIs will be delivered before the region is filled. See ToPA PMI in Section 11.2.4.1. If there is more than one output entry before the END entry, or if the END entry has the wrong base address, an operational error will be signaled (see "ToPA Errors" in Section 11.2.4.1).
	30:2	Reserved	
	31	IP Payloads are LIP	1: Generated packets which contain IP payloads have LIP values, which include the CS base component. 0: Generated packets which contain IP payloads have RIP values, which are the offset from CS base.
EDX	31:0	Reserved	

11.3.1.1 Packet Decoding of RIP versus LIP

FUP, TIP, TIP.PGE, and TIP.PGE packets can contain an IP payload. On some processor generations, this payload will be an effective address (RIP), while on others this will be a linear address (LIP). In the former case, the payload is the offset from the current CS base address, while in the latter it is the sum of the offset and the CS base address. Which IP type is in use is indicated by enumeration (see Table 11-11).

For software that executes while the CS base address is 0 (including all software executing in 64-bit mode), the difference is indistinguishable. A trace decoder must account for cases, where the CS base address is not 0 and the distinction can be seen.

11.3.1.2 Model Specific Capability Restrictions

Some processor generations impose the following restrictions that prevent use of LBRs, BTS, BTM, or LERs when software has enabled tracing with Intel Processor Trace:

- If packet generation is enabled (IA32_RTIT_CTL.TraceEn = 1), any attempt to enable LBRs, LERs, BTS, or BTM (setting IA32_DEBUG_CTL.LBR =1 or IA32_DEBUG_CTL.TR = 1) will cause a general-protection fault (#GP). Further, any read or write of LBRs or LERs will cause a #GP. Enabling packet generation clears the LBRs, LERs, and the LBR TOS pointer.
- If LBR, BTS, or BTM is enabled, any attempt to enable trace packet generation will cause a #GP.
- A RSM that attempts to set both TraceEn and IA32_DEBUGCTL.LBR or IA32_DEBUGCTL.TR will go to shutdown.

For processor with CPUID DisplayFamily_DisplayModel signature of 06_3DH and 06_4AH, the use of Intel PT and LBRs are mutually exclusive.

11.3.2 Enabling and Configuration of Trace Packet Generation

To configure trace packets, enable packet generation, and capture packets, software starts with using CPUID instruction to detect its feature flag, CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1; followed by enumerating the capabilities described in Section 11.3.1.

Based on the capability queried from Section 11.3.1, software must configure a number of model-specific registers. This section describes programming considerations related to those MSRs.

11.3.2.1 Enabling Packet Generation

When configuration and enabling packet generation, the IA32_RTIT_CTL MSR should be written last, since writes to the other configuration MSRs cause a general-protection fault (#GP) if TraceEn = 1. If a prior trace collection context is not being restored, then software should first clear IA32_RTIT_STATUS. This is important since the Stopped, and Error fields are writable; clearing the MSR clears any values that may have persisted from prior trace packet collection contexts. See Section 11.2.5.2 for details of packets generated by setting TraceEn to 1.

If setting TraceEn to 1 causes an operational error (see Section 11.3.7), there may be a delay after the WRMSR completes before the error is signaled in the IA32_RTIT_STATUS MSR.

While packet generation is enabled, the values of some configuration MSRs (e.g., IA32_RTIT_STATUS and IA32_RTIT_OUTPUT_*) are transient, and reads may return values that are out of date. Only after packet generation is disabled (by clearing TraceEn) do reads of these MSRs return reliable values.

11.3.2.2 Disabling Packet Generation

After disabling packet generation by clearing IA32_RTIT_CTL, it is advisable to read the IA32_RTIT_STATUS MSR (Section 11.2.5.3):

- If the Error bit is set, an operational error was encountered, and the trace is most likely compromised. Software should check the source of the error (by examining the output MSR values), correct the source of the problem, and then attempt to gather the trace again. For details on operational errors, see Section 11.3.7. Software should clear IA32_RTIT_STATUS.Error before re-enabling packet generation.
- If the Stopped bit is set, software execution encountered the ToPA Stop condition (see "ToPA STOP" in Section 11.2.4.1) before packet generation was disabled.

11.3.3 Forcing Packet Output to Be Written to Memory

Packets are first buffered internally and then written to memory asynchronously. To collect packet output for post-processing, a collector needs first to ensure that all internally buffered packets have been written to memory. Software can ensure this by stopping packet generation by clearing IA32_RTIT_CTL.TraceEn (see "Disabling Packet Generation" in Section 11.2.5.2).

When this operations complete, the IA32_RTIT_OUTPUT_* MSR values can be read to discover where the trace ended.

11.3.4 Context Switch Consideration

To facilitate construction of instruction execution traces at the granularity of a software process or thread context, software can save and restore the states of the trace configuration MSRs across the process or thread context switch boundary. The principle is the same as saving and restoring the typical architectural processor states across context switches.

The configuration can be saved and restored through a sequence of WRMSR and RDMSR instructions, respectively. To stop tracing and to ensure that all configuration MSRs contain stable values, software must clear IA32_RTIT_CTL.TraceEn before reading any other trace configuration MSRs. The recommended method for saving trace configuration context manually follows:

1. RDMSR IA32_RTIT_CTL, save value to memory
2. WRMSR IA32_RTIT_CTL with saved value from RDMSR above and TraceEn cleared

3. RDMSR all other configuration MSRs whose values had changed from previous saved value, save changed values to memory

When restoring the trace configuration context, IA32_RTIT_CTL should be restored last:

1. Read saved configuration MSR values, aside from IA32_RTIT_CTL, from memory, and restore them with WRMSR
2. Read saved IA32_RTIT_CTL value from memory, and restore with WRMSR.

11.3.5 Decoder Synchronization (PSB+)

The PSB packet (Section 11.4.2.12) serves as a synchronization point for a trace-packet decoder. It is a pattern in the trace log for which the decoder can quickly scan to align packet boundaries. No legal packet combination can result in such a byte sequence. As such, it serves as the starting point for packet decode. To decode a trace log properly, the decoder needs more than simply to be aligned: it needs to know some state and potentially some timing information as well.

When a PSB packet is generated, it is followed by a PSBEND packet (Section 11.4.2.13). One or more packets will be generated in between those two packets, and these inform the decoder of the current state of the processor. These packets, known collectively as PSB+, should be interpreted as "status only", since they do not imply any change of state at the time of the PSB, nor are they associated directly with any instruction or event. Thus, the normal binding and ordering rules that apply to these packets outside of PSB+ can be ignored when these packets are between a PSB and PSBEND. They inform the decoder of the state of the processor at the time of the PSB.

PSB+ can include:

- Timestamp (TSC), if IA32_RTIT_CTL.TSCEn=1
- Paging Info Packet (PIP), if ContextEn=1 and IA32_RTIT_CTL.OS=1
- Core Bus Ratio (CBR)
- MODE, including all supported MODE leaves, if ContextEn=1.
- Flow Update Packet (FUP), if ContextEn=1 The ordering of packets within PSB+ is not guaranteed to match on all processors implementations.

PSB is generated only when TriggerEn=1; hence PSB+ has the same dependencies.

Note that an overflow can occur during PSB+, and this could cause the PSBEND packet to be lost, potentially causing the decoder to treat all subsequent packets as "status only" until the next PSB. For this reason, the OVF packet should also be viewed as terminating PSB+.

11.3.6 Internal Buffer Overflow

In the rare circumstances when new packets need to be generated but the processor's dedicated internal buffers are all full, an "internal buffer overflow" occurs. On such an overflow packet generation ceases (as packets would need to enter the processor's internal buffer) until the overflow resolves. Once resolved packet generation resumes.

The buffer overflow condition might not be cleared until the buffer has been completely written to memory and is empty. When the buffer overflow is cleared naturally, an OVF packet (Section 11.4.2.11) is generated, and the internal state for compressing LIPs or RETs is cleared. This ensures that the next IP will not be compressed against a lost IP packet, and any RETs seen whose CALLs occurred before the overflow will not be compressed.

The OVF packet will be followed by a FUP or TIP.PGE, the payload of which will be the Current IP of the first instruction after the overflow is cleared. Between the OVF and following FUP or TIP.PGE, there may be other packets that are not dependent on ContextEn, even a full PSB+.

The IP in the FUP or TIP.PGE is that of the instruction at which packet generation resumes. Thus, on clearing of a buffer overflow, the decoder will know exactly where the processor is now executing, although it will not know the exact instruction where the buffer overflow occurred.

11.3.7 Operational Errors

Errors are detected as a result of packet output configuration problems, which can include output alignment issues, ToPA reserved bit violations, or overlapping packet output with restricted memory. See "ToPA Errors" in Section 11.2.4.1 for details on ToPA errors, and Section 11.2.4.2 for details on restricted memory errors. Operational errors are only detected and signaled when TraceEn=1.

When an operational error is detected, tracing is disabled and the error is logged. Specifically, IA32_RTIT_STATUS.Error is set, which will cause IA32_RTIT_STATUS.TriggerEn to be 0. This will disable generation of all packets. Some causes of operational errors may lead to packet bytes being dropped.

It should be noted that the timing of error detection may not be predictable. Errors are signaled when the processor encounters the problematic configuration. This could be as soon as packet generation is enabled but could also be later when the problematic entry or field needs to be used.

Once an error is signaled, software should disable packet generation by clearing TraceEn, diagnose and fix the error condition, and clear IA32_RTIT_STATUS.Error. At this point, packet generation can be re-enabled.

11.4 TRACE PACKETS AND DATA TYPES

This section details the data packets generated by Intel Processor Trace. It is useful for developers writing the interpretation code that will decode the data packets and apply it to the traced source code.

11.4.1 Packet Relationships and Ordering

This section introduces the concept of packet "binding", which involves determining the IP in a binary disassembly at which the change indicated by a given packet applies. Some packets have the associated IP as the payload (FUP, TIP), while for others the decoder need only search for the next instance of a particular instruction (or instructions) to bind the packet (TNT). However, in many cases, the decoder will need to consider the relationship between packets, and to use this packet context to determine how to bind the packet.

Section 11.4.2 below provides detailed descriptions of the packets, including how packets bind to IPs in the disassembly, to other packets, or to nothing at all. Many packets listed are simple to bind, because they are generated in only a few scenarios. Those that require more consideration are typically part of "compound packet events", such as interrupts, exceptions, and some instructions, where multiple packets are generated by a single operation (instruction or event). These compound packet events frequently begin with a FUP to indicate the source address (if it is not clear from the disassembly), and are concluded by a TIP or TIP.PGD packet that indicates the destination address (if one is provided). In this scenario, the FUP is said to be "coupled" with the TIP packet.

Other packets could be in between the coupled FUP and TIP packet. When the workload being traced changes CR3 or the processor's mode of execution, ia state update packet (i.e., PIP or MODE) is generated. A summary of compound packet events is provided in Table 11-12; see Section 11.4.2 for more per-packet details and Section 11.4.3 for more detailed packet generation examples.

Table 11-12. Compound Packet Event Summary

Event Type	Beginning	Middle	End	Comment
Control-flow transfer	FUP or none	Any combination of PIP, MODE.Exec, or none	TIP or TIP.PGD	FUP only for asynchronous events. Order of middle packets may vary. PIP /MODE only if the operation modifies the state tracked by these respective packets
TSX Update	MODE.TSX, and (FUP or none)	None	TIP, TIP.PGD, or none	FUP TIP/TIP.PGD only for TSX abort cases
Overflow	OVF	None	FUP or TIP.PGE	FUP if overflow resolves while ContextEn=1, else TIP.PGE.

11.4.2 Packet Definitions

The following description of packet definitions are in tabular format. Figure 11-3 explains how to interpret them.

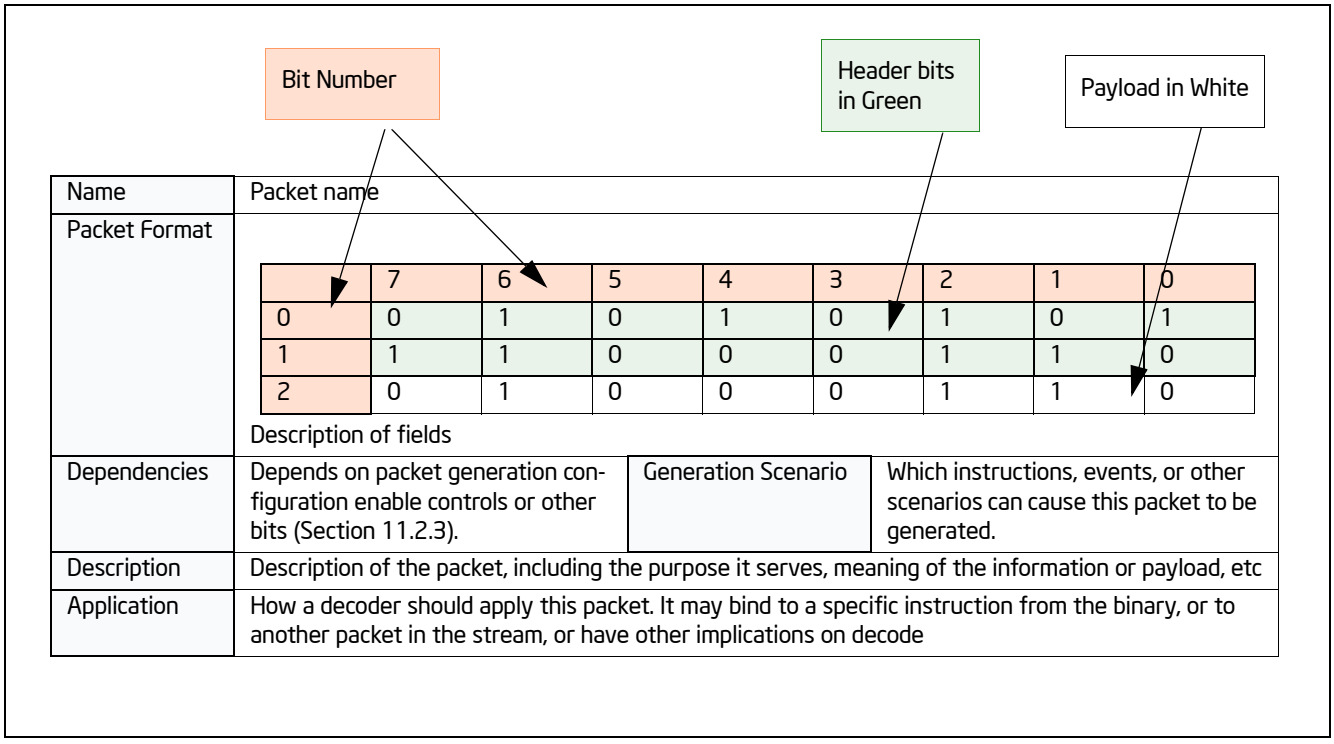


Figure 11-3. Interpreting Tabular Definition of Packet Format

11.4.2.1 Taken/Not-taken (TNT) Packet

Table 11-13. TNT Packet Definition

Name	Taken/Not-taken (TNT) Packet
Packet Format	

	7	6	5	4	3	2	1	0	
0	1	B_1	B_2	B_3	B_4	B_5	B_6	0	Short TNT

B1...BN represent the last N conditional branch or compressed RET (Section 11.4.2.2) results, such that B1 is oldest and BN is youngest. The short TNT packet can contain from 1 to 6 TNT bits. The long TNT packet can contain up from 1 to 47 TNT bits.

	7	6	5	4	3	2	1	0	
0	0	1	0	0	0	0	1	0	Long TNT
1	1	0	1	0	0	0	1	1	
2	B_{40}	B_{41}	B_{42}	B_{43}	B_{44}	B_{45}	B_{46}	B_{47}	
3	B_{32}	B_{33}	B_{34}	B_{35}	B_{36}	B_{37}	B_{38}	B_{39}	
4	B_{24}	B_{25}	B_{26}	B_{27}	B_{28}	B_{29}	B_{30}	B_{31}	
5	B_{16}	B_{17}	B_{18}	B_{19}	B_{20}	B_{21}	B_{22}	B_{23}	
6	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}	
7	1	B_1	B_2	B_3	B_4	B_5	B_6	B_7	

Table 11-13. TNT Packet Definition

<table>
<tr><td></td><td>Irrespective of how many TNT bits is in a packet, the last valid TNT bit is followed by a trailing 1, or Stop bit, as shown above. If the TNT packet is not full (fewer than 6 TNT bits for the Short TNT, or fewer than 47 TNT bits for the Long TNT), the Stop bit moves up, and the trailing bits of the packet are filled with 0s. Examples of these “partial TNTs” are shown below.

<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td></td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>B_1</td><td>B_2</td><td>B_3</td><td>B_4</td><td>0</td><td>Short TNT</td></tr>
</table>
</td></tr>
<tr><td></td><td>
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td></td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>Long TNT</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td></td></tr>
<tr><td>2</td><td>B_{24}</td><td>B_{25}</td><td>B_{26}</td><td>B_{27}</td><td>B_{28}</td><td>B_{29}</td><td>B_{30}</td><td>B_{31}</td><td></td></tr>
<tr><td>3</td><td>B_{16}</td><td>B_{17}</td><td>B_{18}</td><td>B_{19}</td><td>B_{20}</td><td>B_{21}</td><td>B_{22}</td><td>B_{23}</td><td></td></tr>
<tr><td>4</td><td>B_8</td><td>B_9</td><td>B_{10}</td><td>B_{11}</td><td>B_{12}</td><td>B_{13}</td><td>B_{14}</td><td>B_{15}</td><td></td></tr>
<tr><td>5</td><td>1</td><td>B_1</td><td>B_2</td><td>B_3</td><td>B_4</td><td>B_5</td><td>B_6</td><td>B_7</td><td></td></tr>
<tr><td>6</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td></td></tr>
<tr><td>7</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td></td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn</td><td>Generation Scenario</td><td>On a conditional branch or compressed RET, if it fills the TNT. Also, partial TNTs may be generated at any time, as a result of other packets being generated,
or certain micro-architectural conditions occurring, before the TNT is full.</td></tr>
<tr><td>Description</td><td colspan="3">Provides the taken/not-taken results for the last 1–N conditional branches (Jcc, J*CXZ, or LOOP) or compressed RETs (Section 11.4.2.2). The TNT payload bits should be interpreted as follows:
▪ 1 indicates a taken conditional branch, or a compressed RET
▪ 0 indicates a not-taken conditional branch
Note that a full TNT packet that causes a buffer overflow may be delayed instead of being dropped and could be sent out before the buffer overflow packet is sent out</td></tr>
<tr><td>Application</td><td colspan="3">Each valid payload bit (that is, bits between the header bits and the trailing Stop bit) applies to an upcoming conditional branch or RET instruction. Once a decoder consumes a TNT packet with N valid payload bits, these bits should be applied to (and hence provide the destination for) the next N conditional branches or RETs</td></tr>
</table>

11.4.2.2 Target IP (TIP) Packet

T

Table 11-14. IP Packet Definition

Name	Target IP (TIP) Packet

Table 11-14. IP Packet Definition

<table>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td colspan="3">IPBytes</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">TargetIP[7:0]</td></tr>
<tr><td>2</td><td colspan="8">TargetIP[15:8]</td></tr>
<tr><td>3</td><td colspan="8">TargetIP[23:16]</td></tr>
<tr><td>4</td><td colspan="8">TargetIP[31:24]</td></tr>
<tr><td>5</td><td colspan="8">TargetIP[39:32]</td></tr>
<tr><td>6</td><td colspan="8">TargetIP[47:40]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn</td><td>Generation Scenario</td><td>Indirect branch (including uncompressed RET), far branch, interrupt, exception, INIT, SIPI, TSX abort.</td></tr>
<tr><td>Description</td><td colspan="3">Provides the target for some control flow transfers</td></tr>
<tr><td>Application</td><td colspan="3">Anytime a TIP is encountered, it indicates that control was transferred to the IP provided in the payload.</td></tr>
<tr><td></td><td colspan="3">The source of this control flow change, and hence the IP or instruction to which it binds, depends on the packets that precede the TIP. If a TIP is encountered and all preceding packets have already been bound, then the TIP will apply to the upcoming indirect branch, far branch, or RSM. However, if there was a preceding FUP that remains unbound, it will bind to the TIP. Here, the TIP provides the target of an asynchronous event or TSX abort that occurred at the IP given in the FUP payload. Note that there may be other packets, in addition to the FUP, which will bind to the TIP packet. See the packet application descriptions for other packets for details.</td></tr>
</table>

IP Compression

The IP payload in a TIP. FUP, TIP.PGE, or TIP.PGD packet can vary in size, based on the mode of execution, and the use of IP compression. IP compression is an optional compression technique the processor may choose to employ to reduce bandwidth. With IP compression, the IP to be represented in the payload is compared with the last IP sent out, via any of FUP, TIP, TIP.PGE, or TIP.PGD. If that previous IP had the same upper (most significant) address bytes, those matching bytes may be suppressed in the current packet. The processor maintains an internal state of the "Last IP" that was encoded in trace packets, thus the decoder will need to keep track of the "Last IP" state in software, to match fidelity with packets generated by hardware.

The "IPBytes" field of the IP packets (FUP, TIP, TIP.PGE, TIP.PGD) serves to indicate how many bytes of payload are provided, and how the decoder should fill in any suppressed bytes. The algorithm for reconstructing the IP for a TIP/FUP packet is shown in the table below.

Table 11-15. FUP/TIP IP Reconstruction

<table>
<tr><td rowspan="2">IPBytes</td><td colspan="8">Uncompressed IP Value</td></tr>
<tr><td>63:56</td><td>55:48</td><td>47:40</td><td>39:32</td><td>31:24</td><td>23:16</td><td>15:8</td><td>7:0</td></tr>
<tr><td>000b</td><td colspan="8">None, IP is out of context</td></tr>
<tr><td>001b</td><td colspan="7">Last IP[63:16]</td><td>IP Payload[15:0]</td></tr>
<tr><td>010b</td><td colspan="5">Last IP[63:32]</td><td colspan="3">IP Payload[31:0]</td></tr>
<tr><td>011b</td><td colspan="2">IP Payload[47] extended</td><td colspan="6">IP Payload[47:0]</td></tr>
<tr><td>100b</td><td colspan="8">Reserved</td></tr>
<tr><td>101b</td><td colspan="8">Reserved</td></tr>
<tr><td>110b</td><td colspan="8">Reserved</td></tr>
<tr><td>111b</td><td colspan="8">Reserved</td></tr>
</table>

Note that the processor-internal Last IP state may be cleared at any time, but is guaranteed to be cleared when a PSB is sent out. When the internal Last IP is cleared, this means that the next FUP/TIP/TIP.PGE/TIP.PGD will have no IP compression.

At times, "IPbytes" will have a value of 0. As shown above, this does not mean that the IP payload matches the full address of the last IP, but rather that the IP for this packet was suppressed. This is used for cases where the IP that applies to the packet is out of context. An example is the TIP.PGD sent on a SYSCALL, when tracing only USR code. In that case, no TargetIP will be included in the packet, since that would expose an instruction point at CPL = 0. When the IP payload is suppressed in this manner, Last IP is not cleared, and instead refers to the last IP packet with a non-zero IPBytes field.

Indirect Transfer Compression for Returns (RET)

In addition to IP compression, TIP packets for near return (RET) instructions can also be compressed. If the RET target matches the next IP of the corresponding CALL, then the TIP packet is unneeded, since the decoder can deduce the target IP by maintaining a CALL/RET stack of its own.

A CALL/RET stack can be maintained by the decoder by doing the following:

1. Allocate space to store 64 RET targets.
2. For near CALLs, push the Next IP onto the stack. Once the stack is full, new CALLs will force the oldest entry off the end of the stack, such that only the youngest 64 entries are stored. Note that this excludes zero-length CALLs, which are direct near CALLs with displacement zero (to the next IP). These CALLs typically don't have matching RETs.
3. For near RETs, pop the top (youngest) entry off the stack. This will be the target of the RET.

In cases where the RET is compressed, the target is guaranteed to match the value produced in 2) above. If the target is not compressed, a TIP packet will be generated with the RET target, which may differ from 2).

The hardware ensure that packets read by the decoder will always have seen the CALL that corresponds to any compressed RET. The processor will never compress a RET across a PSB, a buffer overflow, or scenario where PacketEn=0. This means that a RET whose corresponding CALL executed while PacketEn=0, or before the last PSB, etc., will not be compressed.

If the CALL/RET stack is manipulated or corrupted by software, and thereby causes a RET to transfer control to a target that is inconsistent with the CALL/RET stack, then the RET will not be compressed, and will produce a TIP packet. This can happen, for example, if software executes a PUSH instruction to push a target onto the stack, and a later RET uses this target.

When a RET is compressed, a Taken indication is added to the TNT buffer. Because it sends no TIP packet, it also does not update the internal Last IP value, and thus the decoder should treat it the same way. If the RET is not compressed, it will generate a TIP packet (just like when RET compression is disabled, via IA32_RTIT_CTL.DisRETC). For processors that employ deferred TIPs (Section 11.4.2.3), an uncompressed RET will not be deferred, and hence will force out any accumulated TNTs or TIPs. This serves to avoid ambiguity, and make clear to the decoder whether the near RET was compressed, and hence a bit in the in-progress TNT should be consumed, or uncompressed, in which case there will be no in-progress TNT and thus a TIP should be consumed.

Note that in the unlikely case that a RET executes in a different execution mode than the associated CALL, the decoder will need to model the same behavior with its CALL stack. For instance, if a CALL executes in 64-bit mode, a 64-bit IP value will be pushed onto the software stack. If the corresponding RET executes in 32-bit mode, then only the lower 32 target bits will be popped off of the stack, which may mean that the RET does not go to the CALL's Next IP. This is architecturally correct behavior, and this RET could be compressed, thus the decoder should match this behavior

11.4.2.3 Deferred TIPs

The processor may opt to defer sending out the TNT when TIPs are generated. Thus, rather than sending a partial TNT followed by a TIP, both packets will be deferred while the TNT accumulates more Jcc/RET results. Any number of TIP packets may be accumulated this way, such that only once the TNT is filled, or once another packet (e.g., FUP) is generated, the TNT will be sent, followed by all the deferred TIP packets, and finally terminated by the other packet(s) that forced out the TNT and TIP packets. Generation of many other packets (see list below) will force out the TNT and any accumulated TIP packets. This is an optional optimization in hardware to reduce the bandwidth consumption, and hence the performance impact, incurred by tracing.

Table 11-16. TNT Examples with Deferred TIPs

Code Flow	Packets, Non-Deferred TIPS	Packets, Deferred TIPS
0x1000 cmp %rcx, 0 0x1004 jnz Foo // not-taken 0x1008 jmp %rdx	TNT(0b0), TIP(0x1308)	
0x1308 cmp %rcx, 1 0x130c jnz Bar // not-taken 0x1310 cmp %rcx, 2 0x1314 jnz Baz // taken 0x1500 cmp %eax, 7 0x1504 jg Exit // not-taken 0x1508 jmp %r15	TNT(0b010), TIP(0x1100)	
0x1100 cmp %rbx, 1 0x1104 jg Start // not-taken 0x1108 add %rcx, %eax 0x110c ... // **an asynchronous Interrupt arrives** INThandler: 0xcc00 pop %rdx	TNT(0b0), FUP(0x110c), TIP(0xcc00)	TNT(0b00100), TIP(0x1308), TIP(0x1100), FUP(0x110c), TIP(0xcc00)

Generation of the following packets may cause a partial TNT (and any accumulated TIPs) to be sent:

- Flow Update Packet (FUP)
- TIP due to uncompressed RET
- TIP.PGE and TIP.PGD
- Paging Information Packet (PIP)
- MODE
- Packet Stream Boundary (PSB)
- Core Bus Ratio (CBR)

As stated above, the processor may opt to send partial TNTs when TIPs are generated as well.

11.4.2.4 Packet Generation Enable (TIP.PGE)

Table 11-17. TIP.PGE Packet Definition

Name	Target IP - Packet Generation Enable (TIP.PGE)							
Packet Format								
	7	6	5	4	3	2	1	0
0	IPBytes			1	0	0	0	1
1	TargetIP[7:0]							
2	TargetIP[15:8]							
3	TargetIP[23:16]							
4	TargetIP[31:24]							
5	TargetIP[39:32]							
6	TargetIP[47:40]							

Table 11-17. TIP.PGE Packet Definition

Dependencies	PacketEn transitions to 1	Generation Scenario	Any branch instruction, control flow transfer, or MOV CR3 that sets PacketEn, a WRMSR that enables packet generation and sets PacketEn
Description	Indicates that PacketEn has transitioned to 1. It provides the IP at which the tracing begins. This can occur due to any of the enables that comprise PacketEn transitioning from 0 to 1, as long as all the others are asserted. Examples ▪ TriggerEn: This is set on software write to set IA32_RTIT_CTL.TraceEn as long as the Stopped and Error bits in IA32_RTIT_STATUS are clear. The IP payload will be the Next IP of the WRMSR. ▪ ContextEn: This is set on a CPL change, a CR3 write. The IP payload will be the Next IP of the instruction that changes context, if it is not a branch, otherwise it will be the target of the branch		
Application	TIP.PGE packets bind to the instruction at the IP given in the payload.		

11.4.2.5 Packet Generation Disable (TIP.PGD)

Table 11-18. TIP.PGD Packet Definition

Name	Target IP - Packet Generation Disable (TIP.PGD)		
Packet Format			

	7	6	5	4	3	2	1	0
0	IPBytes			0	0	0	0	1
1	TargetIP[7:0]							
2	TargetIP[15:8]							
3	TargetIP[23:16]							
4	TargetIP[31:24]							
5	TargetIP[39:32]							
6	TargetIP[47:40]							

Dependencies	PacketEn transitions to 0	Generation Scenario	Any branch instruction, control flow transfer, or MOV CR3 that clears PacketEn, a WRMSR that disables packet generation and clears PacketEn
Description	Indicates that PacketEn has transitioned to 0. It will include the IP at which the tracing ends, unless ContextEn= 0 or TraceEn=0 at the conclusion of the instruction or event that cleared PacketEn. PacketEn can be cleared due to any of the enables that comprise PacketEn transitioning from 1 to 0. Examples: ▪ TriggerEn: This is cleared on software write to clear IA32_RTIT_CTL.TraceEn, or on ToPA STOP, or on operational error. The IP payload will be suppressed in this case, and the "IPBytes" field will have the value 0. ▪ ContextEn: This can happen on a CPL change, or a CR3 write. See Section 11.2.3.3 for details. In this case, when ContextEn is cleared, there will be no IP payload. The "IPBytes" field will have value 0 Note that, in cases where a branch that would normally produce a TIP packet (i.e., far transfer, indirect branch, interrupt, etc) or TNT update (conditional branch or compressed RT) causes PacketEn to transition from 1 to 0, the TIP or TNI bit will be replaced with TIP.PGD.		
Application	TIP.PGD can be produced by any branch instructions, as well as some non-branch instructions, that clear PacketEn. When produced by a branch, it replaces any TIP or TNT update that the branch would normally produce. In cases where there is an unbound FUP preceding the TIP.PGD, then the TIP.PGD is part of compound operation (i.e., asynchronous event or TSX abort) which cleared PacketEn. For most such cases, the TIP.PGD is simply replacing a TIP, and should be treated the same way. If there is not an associated FUP, the binding will depend on whether there is an IP payload. If there is an IP payload, then the TIP.PGD should be applied to either the next direct branch whose target matches the TIP.PGD payload, or the next branch that would normally generate a TIP or TNT packet. If there is no IP payload, then the TIP.PGD should apply to the next branch or MOV CR3 instruction		

11.4.2.6 Flow Update (FUP) Packet

Table 11-19. FUP Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Float Update (FUP) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><th></th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td colspan="3">IPBytes</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">IP[7:0]</td></tr>
<tr><td>2</td><td colspan="8">IP[15:8]</td></tr>
<tr><td>3</td><td colspan="8">IP[23:16]</td></tr>
<tr><td>4</td><td colspan="8">IP[31:24]</td></tr>
<tr><td>5</td><td colspan="8">IP[39:32]</td></tr>
<tr><td>6</td><td colspan="8">IP[47:40]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn</td><td>Generation Scenario</td><td colspan="2">Asynchronous Events (interrupts, exceptions, INIT, SIPI, SMI, #MC), XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, a WRMSR that disables packet generation, PSB+</td></tr>
<tr><td>Description</td><td colspan="4">Provides the source address for asynchronous events, and some other instructions. Is never sent alone, always sent with an associated TIP or MODE packet, and potentially others</td></tr>
<tr><td>Application</td><td colspan="4">FUP packets provide the IP to which they bind. However, they are never standalone, but are coupled with other packets.
In TSX cases, the FUP is immediately preceded by a MODE.TSX, which binds to the same IP. A TIP will follow only in the case of TSX aborts, see Section 11.4.2.8 for details.
Otherwise, FUPs are part of compound packet events (see Section 11.4.1). In these compound cases, the FUP provides the source IP for an instruction or event, while a following TIP (or TIP.PGD) uop will provide any destination IP. Other packets may be included in the compound event between the FUP and TIP.</td></tr>
</table>

FUP IP Payload

Flow Update Packet gives the source address of an instruction when it is needed. In general, branch instructions do not need a FUP, because the source address is clear from the disassembly. For asynchronous events, however, the source address cannot be inferred from the source, and hence a FUP will be sent. Table 11-20 illustrates cases where FUPs are sent, and which IP can be expected in those cases.

Table 11-20. FUP Cases and IP Payload

Event	Flow Update IP	Comment
External Interrupt, NMI/SMI, Traps, Machine Check (trap-like), Software Interrupt, INIT/SIPI	Address of next instruction (Next IP) that would have been executed	Functionally, this matches the LBR FROM field value.
Exceptions/Faults, Machine check (fault-like)	Address of the instruction which took the exception/fault (Current IP)	This matches the similar functionality of LBR FROM field value.
XACQUIRE	Address of the X* instruction	
XRELEASE, XBEGIN, XEND, XABORT, other transactional abort	Current IP	
#SMI	IP that is saved into SMRAM	
WRMSR that clears TraceEn	Current IP	

On a canonical fault due to sequentially fetching an instruction in non-canonical space (as opposed to jumping to non-canonical space), the IP of the fault (and thus the payload of the FUP) will be a non-canonical address. This is consistent with what is pushed on the stack for such faulting cases.

11.4.2.7 Paging Information (PIP) Packet

Table 11-21. PIP Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Paging Information (PIP) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>2</td><td colspan="7">CR3[11:5] or 0</td><td>RSVD</td></tr>
<tr><td>3</td><td colspan="8">CR3[19:12]</td></tr>
<tr><td>4</td><td colspan="8">CR3[27:20]</td></tr>
<tr><td>5</td><td colspan="8">CR3[35:28]</td></tr>
<tr><td>6</td><td colspan="8">CR3[43:36]</td></tr>
<tr><td>7</td><td colspan="8">CR3[51:44]</td></tr>
</table>
The CR3 payload shown includes only the address portion of the CR3 value. For PAE paging, CR3[11:5] are thus included. For other page modes (32-bit and IA-32e paging), these bits are 0.
The CR3 address size, and hence the size of this packet, depend on the enumerated packet physical address size.
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn && ContextEn && IA32_RTIT_CTL.OS</td><td>Generation Scenario</td><td colspan="2">MOV CR3, Task switch, INIT, SIPI, PSB+</td></tr>
<tr><td>Description</td><td colspan="4">This packet holds the CR3 address value. It will be generated on operations that modify CR3:
▪ MOV CR3 operation
▪ Task Switch
▪ INIT and SIPI
PIPs are not generated, despite changes to CR3, on SMI and RSM. This is due to the special behavior on these operations, see Section 11.2.6.2 for details. Note that, for some cases of task switch where CR3 is not modified, no PIP will be produced.
The purpose of the PIP is to indicate to the decoder which application is running, so that it can apply the proper binaries to the linear addresses that are being traced.
The PIP packet contains the new CR3 value when CR3 is written</td></tr>
<tr><td>Application</td><td colspan="4">The purpose of the PIP packet is to help the decoder uniquely identify what software is running at any given time. When a PIP is encountered, a decoder should do the following:
1) If there was a prior unbound FUP (that is, a FUP not preceded by MODE.TSX, and hence pairs with a TIP that has not yet been seen), then this PIP is part of a compound packet event (Section 11.4.1). Find the ending TIP and apply the new CR3 values to the TIP payload IP.
2) Look for the next MOV CR3 or far branch in the disassembly, and apply the new CR3 to the next (or target) IP.
For examples of the packets generated by these flows, see Section 11.4.3</td></tr>
</table>

11.4.2.8 MODE Packets

MODE packets keep the decoder informed of various processor modes about which it needs to know in order to properly manage the packet output, or to properly disassemble the associated binaries. MODE packets include a header and a mode byte, as shown below.

Table 11-22. General Form of MODE Packets

	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	1
1	Leaf ID			Mode				

The MODE Leaf ID indicates which set of mode bits are held in the lower bits.

MODE.Exec Packet

Table 11-23. MODE.Exec Packet Definition

<table>
<tr><td>Name</td><td colspan="4">MODE.Exec Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>CS.D</td><td>(CS.L & LMA)</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn</td><td>Generation Scenario</td><td colspan="2">Far branch, interrupt, exception, PSB+, and any scenario that can generate a TIP.PGE</td></tr>
<tr><td>Description</td><td colspan="4">Indicates whether software is in 16, 32, or 64-bit mode, by providing the CS.D and (CS.L & IA32_EFER.LMA) values. Essential for the decoder to properly disassemble the associated binary.

<table>
<tr><td>CS.D</td><td>(CS.L & IA32_EFER.LMA)</td><td>Addressing Mode</td></tr>
<tr><td>1</td><td>1</td><td>N/A</td></tr>
<tr><td>0</td><td>1</td><td>64-bit mode</td></tr>
<tr><td>1</td><td>0</td><td>32-bit mode</td></tr>
<tr><td>0</td><td>0</td><td>16-bit mode</td></tr>
</table>

For cases where the mode changes while TraceEn=1 but PacketEn=0 (i.e., when packet generation is enabled but software is out of context), and the mode change persists once tracing resumes (once PacketEn=1), the processor will send a MODE.Exec packet preceding the subsequent TIP.PGE. Further, any time packet generation is disabled, if it is re-enabled the first TIP.PGE will be preceded by a MODE.Exec packet. This serves to cover cases where the mode changes while packet generation is disabled.</td></tr>
<tr><td>Application</td><td colspan="4">MODE.Exec always immediately precedes a TIP or TIP.PGE. The mode change applies to the IP address in the payload of the next TIP or TIP.PGE.</td></tr>
</table>

MODE.TSX Packet

Table 11-24. MODE.TSX Packet Definition

<table>
<tr><td>Name</td><td colspan="4">MODE.TSX Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>TXAbort</td><td>InTX</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn and ContextEn</td><td>Generation Scenario</td><td colspan="2">XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, Asynchronous TSX Abort</td></tr>
</table>

Table 11-24. MODE.TSX Packet Definition

<table>
<tr><td>Description</td><td>Indicates when a TSX transaction (either HLE or RTM) begins, commits, or aborts. Instructions executed transactionally will be “rolled back” if the transaction is aborted.

<table>
<tr><th>TXAbort</th><th>InTX</th><th>Implication</th></tr>
<tr><td>1</td><td>1</td><td>N/A</td></tr>
<tr><td>0</td><td>1</td><td>Transaction begins, or executing transactionally</td></tr>
<tr><td>1</td><td>0</td><td>Transaction aborted</td></tr>
<tr><td>0</td><td>0</td><td>Transaction committed, or not executing transactionally</td></tr>
</table>
</td></tr>
<tr><td>Application</td><td>MODE.TSX always immediately precedes a FUP. If the TXAbort bit is zero, then the mode change applies to the IP address in the payload of the FUP. If TXAbort=1, then the FUP will be followed by a TIP, and the mode change will apply to the IP address in the payload of the TIP.</td></tr>
</table>

11.4.2.9 Core:Bus Ratio (CBR) Packet

Table 11-25. CBR Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Core:Bus Ratio (CBR) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><th></th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>2</td><td colspan="8">Core:Bus Ratio</td></tr>
<tr><td>3</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn</td><td>Generation Scenario</td><td colspan="2">Any change to core:bus ratio (frequency change, sleep state wake), PSB+, and after modifying configuration MSR enable</td></tr>
<tr><td>Description</td><td colspan="4">Indicates the core:bus ratio of the processor core. Byte 2 represents the number of core clock cycles per bus clock cycle. Useful for correlating wall-clock time and cycle time</td></tr>
<tr><td>Application</td><td colspan="4">When TSC packets are enabled, a TSC packet will precede the CBR. If there was a core:bus ratio (frequency) change, the TSC payload provides the time at which it occurred.
All packets following the CBR represent instructions that executed with the new core:bus ratio, while all preceding packets (aside from the associated TSC) represent instructions that executed with the prior ratio. There is not a precise IP to which to bind the CBR packet.</td></tr>
</table>

11.4.2.10 Timestamp Counter (TSC) Packet

Table 11-26. TSC Packet Definition

Name	Timestamp Counter (TSC) Packet

Table 11-26. TSC Packet Definition

<table>
<tr><td>Packet Format</td><td colspan="4"><table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">SW TSC[7:0]</td></tr>
<tr><td>2</td><td colspan="8">SW TSC[15:8]</td></tr>
<tr><td>3</td><td colspan="8">SW TSC[23:16]</td></tr>
<tr><td>4</td><td colspan="8">SW TSC[31:24]</td></tr>
<tr><td>5</td><td colspan="8">SW TSC[39:32]</td></tr>
<tr><td>6</td><td colspan="8">SW TSC[47:40]</td></tr>
<tr><td>7</td><td colspan="8">SW TSC[55:48]</td></tr>
</table></td></tr>
<tr><td>Dependencies</td><td>IA32_RTIT_CTL.TSCEn && TriggerEn</td><td>Generation Scenario</td><td>Any change to core:bus ratio (with CBR packet), sleep state wake, STPCLK, PSB+, and on transition of TraceEn from 0 to 1.</td></tr>
<tr><td>Description</td><td colspan="3">When enabled by software, TSC provides the lower 7 bytes of the current TSC value, as returned by the RDTSC instruction. This may be useful for tracking wall-clock time, and synchronizing the packets in the log with other time-stamped logs</td></tr>
<tr><td>Application</td><td colspan="3">TSC packet provides a wall-clock proxy of the event which generated it (packet generation enable, sleep state wake, etc). In all cases, TSC is sent preceding a CBR. TSC does not precisely indicate the time of any control flow packets; however, all preceding packets represent instructions that executed before the indicated TSC time, and all subsequent packets represent instructions that executed after it. There is not a precise IP to which to bind the TSC packet</td></tr>
</table>

11.4.2.11 Overflow (OVF) Packet

Table 11-27. OVF Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Overflow (OVF) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4"><table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
</table></td></tr>
<tr><td>Dependencies</td><td>TriggerEn</td><td>Generation Scenario</td><td colspan="2">On resolution of internal buffer overflow</td></tr>
<tr><td>Description</td><td colspan="4">OVF simply indicates to the decoder that an internal buffer overflow occurred, and packets were likely lost. OVF is followed by a FUP or TIP.PGE which will indicate the point at which packet generation resumes. See Section 11.3.6</td></tr>
<tr><td>Application</td><td colspan="4">When an OVF packet is encountered, the decoder should skip to the IP given in the following FUP or TIP.PGE. Software should reset its call stack depth on overflow, since no RET compression is allowed across an overflow. Similarly, any IP compression that follows the OVF is guaranteed to use as a reference LastIP the IP payload of an IP packet that was not dropped</td></tr>
</table>

11.4.2.12 Packet Stream Boundary (PSB) Packet

Table 11-28. PSB Packet Definition

Name	Packet Stream Boundary (PSB) Packet

Table 11-28. PSB Packet Definition

<table>
<tr><td>Packet Format</td><td colspan="4"><table>
<tr><th></th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>2</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>3</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>4</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>5</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>6</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>7</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>8</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>9</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>10</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>11</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>12</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>13</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>14</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>15</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
</table></td></tr>
<tr><td>Dependencies</td><td>TriggerEn</td><td>Generation Scenario</td><td colspan="2">The frequency of PSB packet generation is implementation specific.</td></tr>
<tr><td>Description</td><td colspan="4">PSB is a unique pattern in the packet output log, and hence serves as a sync point for the decoder. It is a pattern that the decoder can search for in order to get aligned on packet boundaries.
PSB also serves as the leading packet for a set of “status-only” packets collectively known as PSB+ (Section 11.3.5).</td></tr>
<tr><td>Application</td><td colspan="4">When a PSB is seen, the decoder should interpret all following packets as “status only”, until either a PSBEND or OVF packet is encountered. “Status only” implies that the binding and ordering rules to which these packets normally adhere are ignored, and the state they carry can instead be applied to the IP payload in the FUP packet that is included.</td></tr>
</table>

11.4.2.13 PSBEND Packet

Table 11-29. PSBEND Packet Definition

<table>
<tr><td>Name</td><td colspan="4">PSBEND Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4"><table>
<tr><th></th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
</table></td></tr>
<tr><td>Dependencies</td><td>TriggerEn</td><td>Generation Scenario</td><td colspan="2">Always follows PSB packet, separated by PSB+ packets</td></tr>
<tr><td>Description</td><td colspan="4">PSBEND is simply a terminator for the series of “status only” (PSB+) packets that follow PSB (Section 11.3.5).</td></tr>
<tr><td>Application</td><td colspan="4">When a PSBEND packet is seen, the decoder should cease to treat packets as “status only”.</td></tr>
</table>

11.4.2.14 PAD Packet

Table 11-30. PAD Packet Definition

Name	PAD Packet							
Packet Format								
	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0
Dependencies	TriggerEn		Generation Scenario	Implementation specific				
Description	PAD is simply a NOP packet. Processor implementations may choose to add pad packets to improve packet alignment or for implementation-specific reasons.							
Application	Ignore PAD packets							

11.4.3 Packet Generation Scenarios

Table 11-31 illustrates the packets generated in various scenarios. Note that this assumes that TraceEn=1 in IA32_RTIT_CTL, while TriggerEn=1 and Error=0 in IA32_RTIT_STATUS, unless otherwise specified. Entries that do not matter in packet generation are marked "D.C."

Table 11-31. Packet Generation under Different Enable Conditions

Case	Operation	PacketEn Before	PacketEn After	Other Dependencies	Packets Output
1a	Normal non-jump operation	0	0		None
1b	Normal non-jump operation	1	1		None
2b	WRMSR/RSM that changes TraceEn 0 -> 1	0	0	TSC if TSCEn=1	PSB, TSC?, CBR, PSBEND
2c	WRMSR/RSM that changes TraceEn 0 -> 1	0	0		NA
2e	WRMSR/RSM that changes TraceEn 0 -> 1	0	1	TSC if TSCEn=1	PSB, TSC?, PIP(CR3), CBR, MODE.Exec, MODE.TSX, FUP(CLIP), PSBEND, MODE.Exec, TIP.PGE(NLIP)
2h	WRMSR/RSM that changes TraceEn 0 -> 1	1	D.C.		NA
3a	WRMSR that changes TraceEn 1 -> 0	0	0		None
3c	WRMSR that changes TraceEn 1 -> 0	0	1		NA
3b	WRMSR that changes TraceEn 1 -> 0	1	0		FUP(CLIP), TIP.PGD()
3d	WRMSR that changes TraceEn 1 -> 0	1	1		NA
4a	WRMSR that keeps TraceEn=1 (must be same value)	0	0		None
4c	WRMSR that keeps TraceEn=1 (must be same value)	0	1		NA
4d	WRMSR that keeps TraceEn=1 (must be same value)	1	0		NA
4b	WRMSR that keeps TraceEn=1 (must be same value)	1	1		None
5a	MOV to CR3	0	0		None

Table 11-31. Packet Generation under Different Enable Conditions

Case	Operation	PacketEn Before	PacketEn After	Other Dependencies	Packets Output
5b	MOV to CR3	0	1	MODE.Exec if the value is different, or if TraceEn cleared, since last TIP.PGD	PIP(NewCR3), MODE.Exec?, TIP.PGE(NLIP)
5c	MOV to CR3	1	0		TIP.PGD()
5d	MOV to CR3	1	1		PIP(NewCR3)
6a	Unconditional direct near jump	0	0		None
6c	Unconditional direct near jump	0	1		NA
6f	Unconditional direct near jump	1	0		NA
6d	Unconditional direct near jump	1	1		None
7a	Conditional taken jump or compressed RET that does not fill up the internal TNT buffer	0	0		None
7b	Conditional taken jump or compressed RET	0	1		NA
7e	Conditional taken jump or compressed RET	1	0		NA
7c	Conditional taken jump or compressed RET that does not fill up the internal TNT buffer	1	1		None
7d	Conditional taken jump or compressed RET that fills up the internal TNT buffer	1	1		TNT
8a	Conditional non-taken jump	0	0		None
8b	Conditional non-taken jump	0	1		NA
8c	Conditional non-taken jump	1	0		NA
8d	Conditional not-taken jump that fills up the internal TNT buffer	1	1		TNT
9a	Near indirect jump (JMP, CALL, or uncompressed RET)	0	0		None
9b	Near indirect jump (JMP, CALL, or uncompressed RET)	0	1		NA
9c	Near indirect jump (JMP, CALL, or uncompressed RET)	1	0		NA
9d	Near indirect jump (JMP, CALL, or uncompressed RET)	1	1		TIP(BLIP)
10a	Far Branch (CALL/JMP/RET)	0	0		None
10b	Far Branch (CALL/JMP/RET)	0	1	MODE.Exec if the value is different, or if TraceEn cleared, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
10c	Far Branch (CALL/JMP/RET)	1	0		TIP.PGD()

Table 11-31. Packet Generation under Different Enable Conditions

Case	Operation	PacketEn Before	PacketEn After	Other Dependencies	Packets Output
10e	Far Branch (CALL/JMP/RET)	1	1	*PIP if CR3 is updated (i.e., task switch), and OS=1 * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	PIP(NewCR3), MODE.Exec?, TIP(BLIP)
11a	HW Interrupt	0	0		None
11b	HW Interrupt	0	1	* MODE.Exec if the value is different, or if TraceEn cleared, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
11c	HW Interrupt	1	0		FUP(NLIP), TIP.PGD()
11e	HW Interrupt	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	PIP(NewCR3,)?, FUP(NLIP), MODE.Exec?, TIP(BLIP)
12a	SW Interrupt	0	0		None
12b	SW Interrupt	0	1	MODE.Exec if the value is different, or if TraceEn cleared, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
12c	SW Interrupt	1	0		FUP(NLIP), TIP.PGD()
12e	SW Interrupt	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	PIP(NewCR3)?, FUP(NLIP), MODE.Exec?, TIP(BLIP
13a	Exception/Fault	0	0		None
13b	Exception/Fault	0	1	MODE.Exec if the value is different, or if TraceEn cleared, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
13c	Exception/Fault	1	0		FUP(CLIP), TIP.PGD()
13e	Exception/Fault	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	PIP(NewCR3)?, FUP(CLIP), MODE.Exec?, TIP(BLIP)

Table 11-31. Packet Generation under Different Enable Conditions

Case	Operation	PacketEn Before	PacketEn After	Other Dependencies	Packets Output
14a	SMI (TraceEn cleared)	0	0		None
14e	SMI (TraceEn cleared)	0	1		NA
14b	SMI (TraceEn cleared)	1	0		FUP(SMRAM,LIP), TIP.PGD()
14c	SMI (TraceEn cleared)	1	1		NA
15a	RSM, TraceEn restored to 0	0	0		None
15b	RSM, TraceEn restored to 1	0	0		See WRMSR cases for packets on enable
15c	RSM, TraceEn restored to 1	0	1		See WRMSR cases for packets on enable. FUP/TIP.PGE IP is SMRAM.LIP
15d	RSM	1	D.C.		Undefined
16i	Vmext	0	0		None
16a	Vmext	0	1		NA
16f	Vmext	1	0		NA
17a	Vmentry	0	0		None
17d	Vmentry	0	1		NA
17g	Vmentry	1	0		NA
26a	XBEGIN/XACQUIRE	0	0		None
26b	XBEGIN/XACQUIRE	0	1		NA
26c	XBEGIN/XACQUIRE	1	0		NA
26d	XBEGIN/XACQUIRE that does not set InTX	1	1		None
26e	XBEGIN/XACQUIRE that sets InTX	1	1		MODE(InTX=1, TXAbort=0), FUP(CLIP)
27a	XEND/XRELEASE	0	0		None
27b	XEND/XRELEASE	0	1		NA
27c	XEND/XRELEASE	1	0		NA
27d	XEND/XRELEASE that does not clear InTX	1	1		None
27e	XEND/XRELEASE that clears InTX	1	1		MODE(InTX=0, TXAbort=0), FUP(CLIP)
28a	XABORT(Async XAbort, or other)	0	0		None
28b	XABORT(Async XAbort, or other)	0	1		NA
28f	XABORT(Async XAbort, or other)	1	0		NA
28d	XABORT(Async XAbort, or other)	1	1		MODE(InTX=0, TXAbort=1), FUP(CLIP), TIP(BLIP)
29a	PSB threshold reached	0	0	TSC if TSCEn=1	PSB, TSC?, CBR, PSBEND
29c	PSB threshold reached	0	1		NA
29d	PSB threshold reached	1	0		NA
29e	PSB threshold reached	0	0	*TSC if TSCEn=1 * PIP if OS=1	PSB, TSC?, CBR, PIP(CR3)?, MODE.Exec, MODE.TSX, FUP(CLIP), PSBEND

Table 11-31. Packet Generation under Different Enable Conditions

Case	Operation	PacketEn Before	PacketEn After	Other Dependencies	Packets Output
30a	INIT (BSP)	0	0		None
30c	INIT (BSP)	0	1	* MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(ResetLIP)
30d	INIT (BSP)	1	0		FUP(NLIP), TIP.PGD()
30f	INIT (BSP)	1	1	* MODE.Exec if the value is different since last TIP.PGD * PIP if OS=1	FUP(NLIP), PIP(0)?, MODE.Exec?, TIP(ResetLIP)
31a	INIT (AP, goes to wait-for-SIPI)	0	D.C.		None
31b	INIT (AP, goes to wait-for-SIPI)	1	D.C.		FUP(NLIP)
32a	SIPI	0	0		None
32b	SIPI	0	0	* PIP if OS=1	PIP(0)?
32c	SIPI	0	1	* MODE.Exec if the value is different since last TIP.PGD * PIP if OS=1	PIP(0)?, MODE.Exec?, TIP(SipiLIP)
32d	SIPI	1	0		TIP.PGD
32f	SIPI	1	1	* MODE.Exec if the value is different since last TIP.PGD * PIP if OS=1	PIP(0)?, MODE.Exec?, TIP(SipiLIP)
33a	MWAIT (to C0)	D.C.	D.C.		None
33b	MWAIT (to higher C-State)	0	D.C.	TSC if TSCEn=1	TSC?, CBR
33c	MWAIT (to higher C-State)	1	D.C.	TSC if TSCEn=1	TSC?, CBR

11.5 SOFTWARE CONSIDERATIONS

11.5.1 Tracing SMM Code

Nothing prevents an SMM handler from configuring and enabling packet generation for its own use. As described in Section 11.2.6.2, SMI will always clear TraceEn, so the SMM handler would have to set TraceEn in order to enable tracing. There are some unique aspects and guidelines involved with tracing SMM code, which follows:

1. SMM should save away the existing values of any configuration MSRs that SMM intends to modify for tracing. This will allow the non-SMM tracing context to be restored before RSM.
2. It is recommended that SMM wait until it sets CSbase to 0 before enabling packet generation, to avoid possible LIP vs RIP confusion (see Section 11.3.1.1).
3. Packet output cannot be directed to SMRR memory, even while tracing in SMM.
4. Before performing RSM, SMM should take care to restore modified configuration MSRs to the values they had immediately after #SMI. This involves first disabling packet generation by clearing TraceEn, then restoring any other configuration MSRs that were modified.

11.5.2 Cooperative Transition of Multiple Trace Collection Agents

A third-party trace-collection tool should take into consideration the fact that it may be deployed on a processor that supports Intel PT but may run under any operating system.

In such a deployment scenario, Intel recommends that tool agents follow similar principles of cooperative transition of single-use hardware resources, similar to how performance monitoring tools handle performance monitoring hardware:

- Respect the "in-use" ownership of an agent who already configured the trace configuration MSRs (see Section 11.6), where "in-use" can be determined by reading the "enable bits" in the configuration MSRs.
- Relinquish ownership of the trace configuration MSRs by clearing the "enabled bits" of those configuration MSRs.

11.6 ARCHITECTURAL MSRS FOR INSTRUCTION TRACING

Table 11-32. IA-32 Architectural MSRs for Enabling and Configuration of Trace Collection

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Introduced as Architectural MSR
Hex	Decimal			
560H	1376	IA32_RTIT_OUTPUT_BASE	**Trace Output Base Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX[bit 25] = 1)
		6:0	Reserved	
		MAXPHYADDR[1]-1:7	Base physical address of 1st ToPA table.	
		63:MAXPHYADDR	**Reserved.**	
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	**Trace Output Mask Pointers Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX[bit 25] = 1)
		6:0	Reserved	
		31:7	MaskOrTableOffset	
		63:32	**Output Offset.**	
570H	1392	IA32_RTIT_CTL	**Trace Packet Control Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX[bit 25] = 1)
		0	TraceEn	
		1	Reserved, MBZ.	
		2	OS	
		3	User	
		6:4	Reserved, MBZ	
		7	CR3 filter	
		8	ToPA; writing 0 will #GP if also setting TraceEn	
		9	Reserved, MBZ	
		10	TSCEn	
		11	DisRETC	
		12	Reserved, MBZ	

Table 11-32. IA-32 Architectural MSRs(Continued)for Enabling and Configuration of Trace Collection

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Introduced as Architectural MSR
Hex	Decimal			
		13	Reserved; writing 0 will #GP if also setting TraceEn	
		63:14	Reserved, MBZ.	
571H	1393	IA32_RTIT_STATUS	**Tracing Status Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX[bit 25] = 1)
		0	Reserved, writes ignored.	
		1	ContexEn, writes ignored.	
		2	TriggerEn, writes ignored.	
		3	Reserved	
		4	Error (R/W)	
		5	Stopped	
		63:6	**Reserved.**	
572H	1394	IA32_RTIT_CR3_MATCH	**Trace Filter CR3 Match Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX[bit 25] = 1)
		4:0	Reserved	
		63:5	CR3[63:5] value to match	

NOTES:

1. MAXPHYADDR is reported by CPUID.80000008H:EAX[7:0].

INDEX

A

B

C

D

E

F

H

I

L

M

P

R

S

T

U

V

W

X