Knights Corner Instruction Set Reference Manual

July 24, 2012

Reference Number: 327364-001

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

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

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

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

Intel, the Intel® logo, Intel® Pentium®, Intel® Xeon®, Intel® Pentium® 4 Processor, Intel® Core™ Duo, Intel® Core™ 2 Duo, MMX™, Intel® Streaming SIMD Extensions (Intel® SSE), Intel® Advanced Vector Extensions (Intel® AVX) are trademarks or registered trademarks of Intel® Corporation or its subsidiaries in the United States and other countries. *Other names and brands may be claimed as the property of others.

Copyright 2012 Intel® Corporation. All rights reserved.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This document describes new vector instructions for the co-processor code-named Knights Corner.

The major features of the new vector instructions described herein are:

A high performance 64 bit execution environment Knights Corner provides a 64 bit execution environment (see Figure 2.1) similar to that found in the Intel64® Intel® Architecture Software Developer's Manual. Additionally, Knights Corner instructions provides basic support for float64 and int64 logical operations.

32 new vector registers Knights Corner's 64 bit environment offers 32 512-bit wide vector SIMD registers tailored to boost the performance of high performance computing applications. The 512-bit vector SIMD instruction extensions provide comprehensive, native support to handle 32 bit and 64 bit floating-point and integer data, including a rich set of conversions for native data types.

Ternary instructions Most instructions are ternary, with two sources and a different destination. Multiply&add instructions are ternary with three sources, one of which is also the destination.

Vector mask support Knights Corner instructions introduces 8 vector mask registers that allow for conditional execution over the 16 (or 8) elements in a vector instruction, and merging of the results into the destination. Masks allow vectorizing loops that contain conditional statements. Additionally, Knights Corner instructions provides support for updating the value of the vector masks with special vector instructions such as *vcmpmps*.

Coherent memory model The Knights Corner instructions operates in a memory address space that follows the standard defined by the *Intel® 64* achitecture. This feature eases the process of developing vector code.

Gather/Scatter support The Knights Corner instructions features specific gather/scatter instructions that allow manipulation of irregular data patterns of memory (by fetching sparse locations of memory into a dense vector register or vice-versa) thus enabling vectorization of algorithms with complex data structures.

Chapter 2

Instructions Terminology and State

The vector streaming SIMD instruction extensions are designed to enhance the performance of *Intel® 64* processors for scientific and engineering applications.

This chapter introduces Knights Corner instructions terminology and relevant processor state.

2.1 Overview of the Knights Corner instructions Extensions

2.1.1 What are vectors?

The vector is the basic working unit of the Knights Corner instructions. Most instructions use at least one vector. A vector is defined as a sequence of packed data elements. For Knights Corner instructions the size of a vector is 64 bytes. As the support data types are float32, int32, float64 and int64, then a vector consists on either 16 doubleword-size elements or alternatively, 8 quadword-size elements. Only doubleword and quadword elements are supported in Knights Corner instructions.

The number of Knights Corner instructions registers is 32.

Additionally, Knights Corner instructions features *vector masks*. Vector masks allow any set of elements in the destination to be protected from updates during the execution of any operation. A subset of this functionality is the ability to control the vector length of the operation being performed (that is, the span of elements being modified, from the first to the last one); however, it is not necessary that the elements that are modified be consecutive.

2.1.2 Vector mask registers

Most Knights Corner instructions vector instructions use a special extra source, known as the write-mask, sourced from a set of 8 registers called *vector mask registers*. These registers contain one bit for each element that can be held by a regular Knights Corner instructions vector register.

Elements are always either float32, int32, float64 or int64 and the vector size is set to 64 bytes. Therefore, a vector register holds either 8 or 16 elements; accordingly, the length of a vector mask register is 16 bits. For 64 bit datatype instructions, only the 8 least significant bits of the vector mask register are used.

A vector mask register affects an instruction for which it is the write-mask operand at element granularity (either 32 or 64 bits). That means that every element-sized operation and element-sized destination update by a vector instruction is predicated on the corresponding bit of the vector mask register used as the write-mask operand. That has two implications:

- **The instruction's operation is not performed for an element** if the corresponding write-mask bit is not set. This implies that no exception or violation can be caused by an operation on a masked-off element.
- **A destination element is not updated** if the corresponding write-mask bit is not set. Thus, the mask in effect provides a merging behavior for Knights Corner instructions vector register destinations, thereby potentially converting destinations into implicit sources, whenever a write-mask containing any 0-bits is used.

  This merging behavior, and the associated performance hazards, can also occur when writing a vector to memory via a vector store. Vectors are written on a per element basis, based on the vector mask register used as a write-mask. Therefore, no exception or violation can be caused by a write to a masked-off element of a destination vector operand.

  The sticky bits implemented in the MXCSR to indicate that floating-point exceptions occurred, are set based soley upon operations on non-masked vector elements.

The value of a given mask register can be set up as a direct result of a vector comparison instruction, transferred from a GP register, or calculated as a direct result of a logical operation between two masks.

Vector mask registers can be used for purposes other than write-masking. For example, they can be used to to set the EFLAGS based on the 0/0xFFFF/other status of the OR of two vector mask registers. A number of the Knights Corner instructions are provided to support such uses of the vector mask register.

2.1.2.1 Vector mask $k0$

The only exception to the vector mask rules described above is mask $k0$. Mask $k0$ cannot be selected as the write-mask for a vector operation; the encoding that would be expected to select mask $k0$ instead selects an implicit mask of 0xFFFF, thereby effectively disabling masking. Vector mask $k0$ can still be used as any non-write-mask operand for any instruction that takes vector mask operands; it just can't ever be selected as a write-mask.

2.1.2.2 Example of use

Here's an example of a masked vector operation.

The initial state of vector registers zmm0, zmm1, and zmm2 is:

```
        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)
```

Given this state, we will execute the following instruction:

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

The `vpaddd` instruction adds vector elements of 32 bit integers. Since elements are not operated upon when the corresponding bit of the mask is not set, the temporary result would be:

```
      [ ********** ********** 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 temporary result is then written into the destination vector register, zmm2, using vector mask register k3 as the write-mask, 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 write-mask operation on each one of the 8 elements of the source/destination vectors.

2.1.3 Understanding Knights Corner instructions

Knights Corner instructions can be classified depending on the nature of their operands. The majority of the Knights Corner instructions operate on vector registers, with a vector mask register serving as a write-mask. However, in most cases these instructions may have one of the vector source operands stored in either memory or a vector register, and may additionally have one or more non-vector (scalar) operands, such as a *Intel® 64* general purpose register or an immediate value. Additionally, some instructions use vector mask registers as destinations and/or explicit sources. Finally, Knights Corner instructions adds some new scalar instructions.

From the point of view of instruction formats, there are four main types of Knights Corner instructions:

- Vector Instructions
- Vector Memory Instructions
- Vector Mask Instructions
- New Scalar Instructions

2.1.3.1 Knights Corner instructions Vector Instructions

Vector instructions operate on vectors that are sourced from either registers or memory and that can be modified prior to the operation via predefined swizzle and convert functions. The destination is usually a vector register, though some vector instructions may have a vector mask register as either a second destination or the primary destination.

All these instructions work in an element-wise manner: the first element of the first source vector is operated on together with the first element of the second source vector, and the result is stored in the first element of the destination vector, and so on for the remaining 15 (or 7) elements.

As described above, the vector mask() register that serves as the write-mask for a vector instruction determines which element locations are actually operated upon; the mask can disable the operation and update for any combination of element locations.

Most vector instructions have three different vector operands (typically, two sources and one destination) except those instructions that have a single source and thus use only two operands. Additionally, most vector instructions feature an extra operand in the form of the vector mask() register that serves as the write-mask. Thus, we can categorize Knights Corner instructions vector instructions based on the number of vector sources they use:

Vector-Converted Vector/Memory. Vector-converted vector/memory instructions, such as vaddps (which adds two vectors), are ternary operations that take two different sources, a vector register and a converted vector/memory operand, and a separate destination vector register, as follows:

```
zmm0 <= OP(zmm1, S(zmm2, m))
```

where zmm1 is a vector operand that is used as the first source for the instruction, $S(zmm2, m)$ is a converted vector/memory operand that is used as the second source for the instruction, and the result of performing operation OP on the two source operands is written to vector destination register $zmm0$.

A converted vector/memory operand is a source vector operand that it is obtained through the process of applying a *swizzle/conversion function* to either a Knights Corner instructions vector or a memory operand. The details of the swizzle/conversion function are found in section 2.2; note that its behavior varies depending on whether the operand is a register or a memory location, and, for memory operands, on whether the instruction performs a floating-point or integer operation. Each source memory operand must have an address that is aligned to the number of bytes of memory actually accessed by the operand (that is, before the swizzle/convert is performed); otherwise, a #GP fault will result.

Converted Vector/Memory. Converted vector/memory instructions, such as vcvtpu2ps (which converts a vector of unsigned integers to a vector of floats), are binary operations that take a single vector source, as follows:

```
zmm0 <= OP(S(zmm1, m))
```

Vector-Vector-Converted Vector/Memory. Vector-vector-converted vector/memory instructions, of which vfmadd*ps (multiply-add of three vectors) is a good example, are similar to the *vector-converted vector/memory* family of instructions; here, however, the destination vector register is used as a third source as well:

```
zmm0 <= OP(zmm0, zmm1, S(zmm2, m))
```

2.1.3.2 Knights Corner instructions Vector Memory Instructions:

Vector Memory Instructions perform vector loads from and vector stores to memory, with extended conversion support.

As with regular vector instructions, vector memory instructions transfer data from/to memory in an element-wise fashion, with the elements that are actually transferred dictated by the contents of the vector mask that is selected as the write-mask.

There are two basic groups of Knights Corner instructions vector memory instructions, vector loads/broadcasts and vector stores.

Vector Loads/Broadcasts. A vector load/broadcast reads a memory source, performs a predefined *load conversion* function, and replicates the result (in the case of broadcasts) to form a 64-byte 16-element vector (or 8-element for 64 bit datatypes). This vector is then conditionally written element-wise to the vector destination register, with the writes enabled or disabled according to the corresponding bits of the vector mask register selected as the write-mask.

The size of the memory operand is a function of the type of conversion and the number of replications to be performed on the memory operand. We call this special memory operand an *up-converted memory operand*. Each source memory operand must have an address that is aligned to the number of bytes of memory actually accessed by the operand (that is, before the swizzle/convert is performed); otherwise, a #GP fault will result.

A Vector Load operates as follows:

```
zmm0 <= U(m)
```

where $U(m)$ is an up-converted memory operand whose contents are replicated and written to destination register $zmm0$. The mnemonic dictates the degree of replication and the conversion table.

A special sub-case of these instructions are **Vector Gathers**. **Vector Gathers** are a special form of vector loads where, instead of a consecutive chunks of memory, we load a sparse set of memory operands (as many as the vector elements of the destination). Every one of those memory operands must obey the alignment rules; otherwise, a #GP fault will result if the related write-mask bit is not disabled (set to 0).

A Vector Gather operates as follows:

```
zmm0 <= U(mv)
```

where $U(mv)$ is a set of up-converted memory operands described by a base address, a vector of indices and an immediate scale to apply for each index. Every one of those operands is conditionally written to destination vector $zmm0$ (based on the value of the write-mask).

Vector Stores. A vector store reads a vector register source, performs a predefined *store conversion* function, and writes the result to the destination memory location on a per-element basis, with the writes enabled or disabled according to the corresponding bits of the vector mask register selected as the write-mask.

The size of the memory destination is a function of the type of conversion associated with the mnemonic. We call this special memory operand a *down-converted memory operand*. Each memory destination operand must have an address that is aligned to the number of bytes of memory accessed by the operand (pre-conversion, if conversion is performed); otherwise, a #GP fault will result.

A Vector Store operates as follows:

```
m <= D(zmm0)
```

where $zmm0$ is the vector register source whose full contents are down-converted (denoted by $D()$), and written to memory.

A special sub-case of these instructions are **Vector Scatters**. **Vector Scatters** are a special form of vector stores where, instead of writing the source vector into a consecutive chuck of memory, we store each vector element into a different memory location. Every one of those memory destinations must obey the alignment rules; otherwise, a #GP fault will result if the related write-mask bit is not disabled (set to 0).

A Vector Scatter operates as follows:

```
mv <= D(zmm0)
```

where $zmm0$ is the vector register source whose full or partial contents are down-converted (denoted by $D()$), and written to the set of memory locations mv, specified by a base address, a vector of indices and an immediate scale which is applied to every index. Every one of those down-converted elements are conditionally stored in the memory locations based on the value of the write-mask.

2.1.3.3 Knights Corner instructions vector mask Instructions

Vector mask instructions allow programmers to set, copy, or operate on the contents of a given vector mask.

There are three types of vector mask instructions:

- **Mask read/write instructions:** These instruction move data between a general-purpose integer register and a vector mask register, or between two vector mask registers.
- **Flag instructions:** This category, consisting of instructions that modify EFLAGS based on vector mask registers, actually contains only one instruction, kortest.
- **Mask logical instructions:** These instructions perform standard bitwise logical operations between vector mask registers.

2.1.3.4 Knights Corner instructions New Scalar Instructions

In addition to vector, vector memory, and vector mask instructions, Knights Corner instructions adds a few scalar instructions as well. These instructions are useful for increasing the performance of some critical algorithms;

for example, any code that suffers reduced performance due to cache-miss latency can benefit from the new prefetch instructions.

2.2 Knights Corner instructions Swizzles and Converts

Data transformation, in the form of certain data conversions or element rearrangements (for loads, both at once) of one operand, can be performed for free as part of most Knights Corner instructions vector instructions.

Three sorts of data transformations are available:

- **Data Conversions:** Sources from memory can be converted to either 32 bit signed or unsigned integer or 32 bit floating-point before being used. Supported data types in memory are float16, sint8, uint8, sint16, and uint16 for load-op instructions
- **Broadcast:** If the source memory operand contains fewer than the total number of elements, it can be broadcast (repeated) to form the full number of elements of the effective source operand (16 for 32 bit instructions, 8 for 64 bit instructions). Broadcast can be combined with load-type conversions only; load-op instructions can do one or the other: either broadcast, or swizzle and/or up-conversion. There are two broadcast granularities:
  - **1-element granularity** where the 1 element of the source memory operand are broadcast 16 times to form a full 16-element effective source operand (for 32 bit instructions), or 8 times to form a full 8-element effective source operand (for 64 bit instructions).
  - **4-element granularity** where the 4 elements of the source memory operand is broadcast 4 times to form a full 16-element effective source operand (for 32 bit instructions), or 2 times to form a full 8-element effective source operand (for 64 bit instructions).

  Broadcast is very useful for instructions that mix vector and scalar sources, where one of the sources is common across the different operations.
- **Swizzles:** Sources from registers can undergo swizzle transformations (that is, they can be permuted), although only 8 swizzles are available, all of which are limited to permuting within 4-element sets (either of 32 bits or 64 bits each).

Knights Corner instructions also introduces the concept of **Rounding Mode Override** or **Static (per instruction) Rounding Mode**, which efficiently supports the feature of determining the rounding mode for arithmetic operations on a per-instruction basis. Thus one can choose the rounding mode without having to perform costly MXCSR save-modify-restore operations.

Knights Corner extends the swizzle functionality for register-register operands in order to provide rounding mode override capabilities for Knights Corner floating-point instructions instead of obeying the MXCSR.RC bits. All four rounding modes are available via swizzle attribute: Round-up, Round-down, Round-toward-zero and Round-to-nearest. The option is not available for instructions with memory operands. On top of these options, Knights Corner introduces the SAE (suppress-all-exceptions) attribute feature. An instruction with SAE set will not raise any kind of floating-point exception flags, independent of the inputs.

In addition to those transformations, all Knights Corner instructions memory operands may have a special attribute, called the *EH* hint (eviction hint), that indicates to the processor that the data is non-temporal - that is, it is unlikely to be reused soon enough to benefit from caching in the 1st-level cache and should be given priority for eviction. This is, however, a hint, and the processor may implement it in any way it chooses, including ignoring the hint entirely.

Table 2.1 shows the assembly language syntax used to indicate the presence or absence of the EH hint.

B1	Function	Usage	Comment
0		[eax]	(no effect) regular memory operand
1	EH	[eax]{eh}	memory operand with Non-Temporal (Eviction) hint

Table 2.1: **EH** attribute syntax.

Data transformations can only be performed on one source operand at most; for instructions that take two or three source operands, the other operands are always used unmodified, exactly as they're stored in their source registers. In no case do any of the Knights Corner instructions allow using data conversion and swizzling at the same time. Broadcasts, on the other hand, can be combined with data conversions when performing vector loads.

Not all instructions can use all of the different data transformations. Load-op instructions (such as vector arithmetic instructions), vector loads, and vector stores have different data transformation capabilities. We can categorize these transformation capabilities into three families:

- **Load-Op SwizzUpConv:** For a register source, swizzle; for a memory operand, either: (a) broadcast, or (b) convert to 32 bit floats or 32 bit signed or unsigned integers. This is used by vector arithmetic instructions and other load-op instructions. There are two versions, one for 32 bit floating-point instructions and another for 32 bit integer instructions; in addition, the available data transformations differ for register and memory operands.
- **Load UpConv:** Convert from a memory operand to 32 bit floats or 32 bit signed or unsigned integers; used by vector loads and broadcast instructions. For 32 bit floats, there are three different conversion tables based on three different input types. See Section 2.2.2, Load UpConvert.
  There is no load conversion support for 64 bit datatypes.
- **DownConv:** Convert from 32 bit floats or 32 bit signed or unsigned integers to a memory operand; used by vector stores. For 32 bit floats, there are three different conversion tables based on three different output types. See Section 2.2.3, Down-Conversion.
  There is no store conversion support for 64 bit datatypes.

2.2.1 Load-Op Swizzle/Convert

Vector load-op instructions can swizzle, broadcast, or convert one of the sources; we will refer to this as the *swizzle/convert* source, and we will use SwizzUpConv to describe the swizzle/convert function itself. The available SwizzUpConv transformations vary depending on whether the operand is memory or a register, and also in the case of conversions from memory depending on whether the vector instruction is 32 bit integer, 32 bit floating-point, 64 bit integer or 64 bit floating-point. 3 bits are used to select among the different options, so eight options are available in each case.

When the swizzle/convert source is a register, SwizzUpConv allows the choice of one of eight swizzle primitives (one of the eight being the identity swizzle). These swizzle functions work on either 4-byte or 8-byte elements within 16-byte/32-byte boundaries. For 32 bit instructions, that means certain permutations of each set of four elements (16 bytes) are supported, replicated across the four sets of four elements. When the swizzle/convert source is a register, the functionality is the same for both integer and floating-point 32 bit instructions. Table 2.2 shows the available register-source swizzle primitives.

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element across 4-element packets	zmm0 {aaaa}
101	broadcast b element across 4-element packets	zmm0 {bbbb}
110	broadcast c element across 4-element packets	zmm0 {cccc}
111	broadcast d element across 4-element packets	zmm0 {dddd}

Table 2.2: **32 bit Register SwizzUpConv** swizzle primitives. Notation: dcba denotes the 32 bit elements that form one 128-bit block in the source (with 'a' least significant and 'd' most significant), so aaaa means that the least significant element of the 128-bit block in the source is replicated to all four elements of the same 128-bit block in the destination; the depicted pattern is then repeated for all four 128-bit blocks in the source and destination. We use 'ponm lkji hgfe dcba' to denote a full Knights Corner instructions source register, where 'a' is the least significant element and 'p' is the most significant element. However, since each 128-bit block performs the same permutation for register swizzles, we only show the least significant block here. Note that in this table as well as in subsequent ones from this chapter $S_2S_1S_0$ are bits 6-4 from MVEX prefix encoding (see Figure 3.3

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element across 4-element packets	zmm0 {aaaa}
101	broadcast b element across 4-element packets	zmm0 {bbbb}
110	broadcast c element across 4-element packets	zmm0 {cccc}
111	broadcast d element across 4-element packets	zmm0 {dddd}

Table 2.3: **64 bit Register SwizzUpConv** swizzle primitives. Notation: dcba denotes the 64 bit elements that form one 256-bit block in the source (with 'a' least significant and 'd' most significant), so aaaa means that the least significant element of the 256-bit block in the source is replicated to all four elements of the same 256-bit block in the destination; the depicted pattern is then repeated for the two 256-bit blocks in the source and destination. We use 'hgfe dcba' to denote a full Knights Corner instructions source register, where 'a' is the least significant element and 'h' is the most significant element. However, since each 256-bit block performs the same permutation for register swizzles, we only show the least significant block here.

For 64 bit instructions, that means certain permutations of each set of four elements (32 bytes) are supported, replicated across the two sets of four elements. When the swizzle/convert source is a register, the functionality is the same for both integer and floating-point 64 bit instructions. Table 2.3 shows the available register-source swizzle primitives.

When the source is a memory location, load-op swizzle/convert can perform either no transformation, 2 different broadcasts, or four data conversions. Vector load-op instructions cannot both broadcast and perform data conversion at the same time. The conversions available differ depending on whether the associated vector instruction is integer or floating-point, and whether the natural data type is 32 bit or 64 bit. (Note however that there are no load conversions for 64 bit destination data types.)

Source memory operands may have sizes smaller than 64 bytes, expanding to the full 64 bytes of a vector source by means of either broadcasting (replication) or data conversion.

Each source memory operand must have an address that is aligned to the number of bytes of memory actually accessed by the operand (that is, before conversion or broadcast is performed); otherwise, a #GP fault will result. Thus, for SwizzUpConv, any of 4-byte, 16-byte, 32-byte, or 64-byte alignment may be required.

.

$S_2S_1S_0$	Function:	Usage
000	no conversion	[rax]
001	broadcast 1 element (x16)	[rax] {1to16}
010	broadcast 4 elements (x4)	[rax] {4to16}
011	float16 to float32	[rax] {float16}
100	uint8 to float32	[rax] {uint8}
101	reserved	N/A
110	uint16 to float32	[rax] {uint16}
111	sint16 to float32	[rax] {sint16 }

Table 2.4: **32 bit Floating-point Load-op SwizzUpConv**$_{f32}$ swizzle/conversion primitives. We use 'ponm lkji hgfe dcba' to denote a full Knights Corner instructions source register, with each letter referring to a 32 bit element, where 'a' is the least significant element and 'p' is the most significant element. So, for example, 'dcba dcba dcba dcba' shows that the source elements are copied to the destination by replicating the lower 128 bits of the source (the four least significant elements) to each 128-bit block of the destination.

$S_2S_1S_0$	Function:	Usage
000	no conversion	[rax] {16to16} or [rax]
001	broadcast 1 element (x16)	[rax] {1to16}
010	broadcast 4 elements (x4)	[rax] {4to16}
011	reserved	N/A
100	uint8 to uint32	[rax] {uint8}
101	sint8 to sint32	[rax] {sint8}
110	uint16 to uint32	[rax] {uint16}
111	sint16 to sint32	[rax] {sint16 }

Table 2.5: **32 bit Integer Load-op SwizzUpConv**$_{i32}$ (Doubleword) swizzle/conversion primitives. We use 'ponm lkji hgfe dcba' to denote a full Knights Corner instructions source register, with each letter referring to a 32 bit element, where 'a' is the least significant element and 'p' is the most significant element. So, for example, 'dcba dcba dcba dcba' shows that the source elements are copied to the destination by replicating the lower 128 bits of the source (the four least significant elements) to each 128-bit block of the destination.

Table 2.4 shows the available 32 bit floating-point swizzle primitives.

SwizzUpConv conversions to float32s are exact.

Table 2.5 shows the available 32 bit integer swizzle primitives.

Table 2.6 shows the available 64 bit floating-point swizzle primitives.

Finally, Table 2.7 shows the available 64 bit integer swizzle primitives.

2.2.2 Load Up-convert

Vector load/broadcast instructions can perform a wide array of data conversions on the data being read from memory, and can additionally broadcast (replicate) that data across the elements of the destination vector register depending on the instructions. The type of broadcast depends on the opcode/mnemonic being used. We

.

$S_2S_1S_0$	Function:	Usage
000	no conversion	[rax] {8to8} or [rax]
001	broadcast 1 element (x8)	[rax] {1to8}
010	broadcast 4 elements (x2)	[rax] {4to8}
011	reserved	N/A
100	reserved	N/A
101	reserved	N/A
110	reserved	N/A
111	reserved	N/A

Table 2.6: **64 bit Floating-point Load-op SwizzUpConv**$_{f64}$ swizzle/conversion primitives. We use 'hgfe dcba' to denote a full Knights Corner instructions source register, with each letter referring to a 64 bit element, where 'a' is the least significant element and 'h' is the most significant element. So, for example, 'dcba dcba' shows that the source elements are copied to the destination by replicating the lower 256 bits of the source (the four least significant elements) to each 256-bit block of the destination.

.

$S_2S_1S_0$	Function:	Usage
000	no conversion	[rax] {8to8} or [rax]
001	broadcast 1 element (x8)	[rax] {1to8}
010	broadcast 4 elements (x2)	[rax] {4to8}
011	reserved	N/A
100	reserved	N/A
101	reserved	N/A
110	reserved	N/A
111	reserved	N/A

Table 2.7: **64 bit Integer Load-op SwizzUpConv**$_{i64}$ (Quadword) swizzle/conversion primitives. We use 'hgfe dcba' to denote a full Knights Corner instructions source register, with each letter referring to a 64 bit element, where 'a' is the least significant element and 'h' is the most significant element. So, for example, 'dcba dcba' shows that the source elements are copied to the destination by replicating the lower 256 bits of the source (the four least significant elements) to each 256-bit block of the destination.

will refer to this conversion process as up-conversion, and we will use UpConv to describe the load conversion function itself.

Based on that, load instructions could be divided into the following categories:

- *regular loads*: load 16 elements (32 bits) or 8 elements (64 bits), convert them and write into the destination vector
- *broadcast 4-elements*: load 4 elements, convert them (possible only for 32 bit data types), replicate them four times (32 bits) or two times (64 bits) and write into the destination vector
- *broadcast 1-element*: load 1 element, convert it (possible only for 32 bit data types), replicate it 16 times (32 bits) or 8 times (64 bits) and write into the destination vector

Therefore, unlike load-op swizzle/conversion, Load UpConv can perform both data conversion and broadcast simultaneously. We will refer to this process as *up-conversion*, and we will use Load UpConv to describe the load conversion function itself.

When a *broadcast 1-element* is selected, the memory data, after data conversion, has a size of 4 bytes, and is broadcast 16 times across all 16 elements of the destination vector register. In other words, one vector element is fetched from memory, converted to a 32 bit float or integer, and replicated to all 16 elements of the destination register. Using the notation where the contents of the source register are denoted {ponm lkji hgfe dcba}, with each letter referring to a 32 bit element ('a' being the least significant element and 'p' being the most significant element), the source elements map to the destination register as follows:

```
{aaaa aaaa aaaa aaaa}
```

When *broadcast 4-element* is selected, the memory data, after data conversion, has a size of 16 bytes, and is broadcast 4 times across the four 128-bit sets of the destination vector register. In other words, four vector elements are fetched from memory, converted to four 32 bit floats or integers, and replicated to all four 4-element sets in the destination register. For this broadcast, the source elements map to the destination register as follows:

```
{dcba dcba dcba dcba}
```

Table 2.8 shows the different 32 bit Load up-conversion instructions in function of the broadcast function and the conversion datatype. Similarly, Table 2.10 shows the different 64 bit Load up-conversion instructions in function of the broadcast function and datatype.

Datatype	Load (16-element)	Broadcast 4-element	Broadcast 1-element
INT32 (d)	VMOVDQA32	VBROADCASTI32X4	VPBROADCASTD
FP32 (ps)	VMOVAPS	VBROADCASTF32X4	VBROADCASTSS

Table 2.8: **32 bit Load UpConv** load/broadcast instructions per datatype. Elements may be 1, 2, or 4 bytes in memory prior to data conversion, after which they are always 4 bytes. We use 'ponm lkji hgfe dcba' to denote a full Knights Corner instructions source register, with each letter referring to a 32 bit element, where 'a' is the least significant element and 'p' is the most significant element. So, for example, 'dcba dcba dcba dcba' shows that the source elements are copied to the destination by replicating the lower 128 bits of the source (the four least significant elements) to each 128-bit block of the destination.

As with SwizzUpConv, UpConv may have source memory operands with sizes smaller than 64-bytes, which are expanded to a full 64-byte vector by means of broadcast and/or data conversion. Each source memory operand must have an address that is aligned to the number of bytes of memory actually accessed by the operand (that is, before conversion or broadcast is performed); otherwise, a #GP fault will result. Thus, any of 1-byte, 2-byte, 4-byte, 8-byte, 16-byte, 32-byte, or 64-byte alignment may be required.

Table 2.9 shows the available data conversion primitives for 32 bit Load UpConv and for the different datatypes supported.

Table 2.11 shows the 64 bit counterpart of Load UpConv. As shown, no 64 bit conversions are available but the pure "no-conversion" option.

2.2.3 Down-Conversion

Vector store instructions can perform a wide variety of data conversions to the data on the way to memory. We will refer to this process as *down-conversion*, and we will use DownConv to describe the store conversion function itself.

DownConv may have destination memory operands with sizes smaller than 64 bytes, as a result of data conversion. Each destination memory operand must have an address that is aligned to the number of bytes of memory actually accessed by the operand (that is, after data conversion is performed); otherwise, a #GP fault will result.

UpConv$_{i32}$ (INT32)

$S_2S_1S_0$	Function:	Usage
000	no conversion	[rax]
001	reserved	N/A
010	reserved	N/A
011	reserved	N/A
100	uint8 to uint32	[rax] {uint8}
101	sint8 to sint32	[rax] {sint8}
110	uint16 to uint32	[rax] {uint16}
111	sint16 to sint32	[rax] {sint16 }

UpConv$_{f32}$ (FP32)

$S_2S_1S_0$	Function:	Usage
000	no conversion	[rax]
001	reserved	N/A
010	reserved	N/A
011	float16 to float32	[rax] {float16}
100	uint8 to float32	[rax] {uint8}
101	sint8 to float32	[rax] {sint8}
110	uint16 to float32	[rax] {uint16}
111	sint16 to float32	[rax] {sint16 }

Table 2.9: **32 bit Load UpConv** conversion primitives.

Datatype	Load	Broadcast 4-element	Broadcast 1-element
INT64 (q)	VMOVDQA64	VBROADCASTI64X4	VPBROADCASTQ
FP64 (pd)	VMOVAPD	VBROADCASTF64X4	VBROADCASTSD

Table 2.10: **64 bit Load UpConv** load/broadcast instructions per datatype. Elements are always 8 bytes. We use 'hgfe dcba' to denote a full Knights Corner instructions source register, with each letter referring to a 64 bit element, where 'a' is the least significant element and 'h' is the most significant element. So, for example, 'dcba dcba' shows that the source elements are copied to the destination by replicating the lower 256 bits of the source (the four least significant elements) to each 256-bit block of the destination.

Thus, any of 1-byte, 2-byte, 4-byte, 8-byte, 16-byte, 32-byte, or 64-byte alignment may be required.

Table 2.12 shows the available data conversion primitives for 32 bit DownConv and for the different supported datatypes.

Table 2.13 shows the 64 bit counterpart of DownConv. As shown, no 64 bit conversions are available but the pure "no-conversion" option.

UpConv$_{i64}$ (INT64)

$S_2S_1S_0$	Function:	Usage
000	no conversion	[rax] {8to8} or [rax]
001	reserved	N/A
010	reserved	N/A
011	reserved	N/A
100	reserved	N/A
101	reserved	N/A
110	reserved	N/A
111	reserved	N/A

UpConv$_{f64}$ (FP64)

$S_2S_1S_0$	Function:	Usage
000	no conversion	[rax] {8to8} or [rax]
001	reserved	N/A
010	reserved	N/A
011	reserved	N/A
100	reserved	N/A
101	reserved	N/A
110	reserved	N/A
111	reserved	N/A

Table 2.11: **64 bit Load UpConv** conversion primitives.

DownConv$_{i32}$ (INT32)

$S_2S_1S_0$	Function:	Usage
000	no conversion	zmm1
001	reserved	N/A
010	reserved	N/A
011	reserved	N/A
100	uint32 to uint8	zmm1 {uint8}
101	sint32 to sint8	zmm1 {sint8}
110	uint32 to uint16	zmm1 {uint16}
111	sint32 to sint16	zmm1 {sint16 }

DownConv$_{f32}$ (FP32)

$S_2S_1S_0$	Function:	Usage
000	no conversion	zmm1
001	reserved	N/A
010	reserved	N/A
011	float32 to float16	zmm1 {float16}
100	float32 to uint8	zmm1 {uint8}
101	float32 to sint8	zmm1 {sint8}
110	float32 to uint16	zmm1 {uint16}
111	float32 to sint16	zmm1 {sint16 }

Table 2.12: **32 bit DownConv** conversion primitives. Unless otherwise noted, all conversions from floating-point use MXCSR.RC

DownConv$_{i64}$ (INT64)

$S_2S_1S_0$	Function:	Usage
000	no conversion	zmm1
001	reserved	N/A
010	reserved	N/A
011	reserved	N/A
100	reserved	N/A
101	reserved	N/A
110	reserved	N/A
111	reserved	N/A

DownConv$_{f64}$ (FP64)

$S_2S_1S_0$	Function:	Usage
000	no conversion	zmm1
001	reserved	N/A
010	reserved	N/A
011	reserved	N/A
100	reserved	N/A
101	reserved	N/A
110	reserved	N/A
111	reserved	N/A

Table 2.13: **64 bit DownConv** conversion primitives.

2.3 Static Rounding Mode

As described before, Knights Corner introduces a new instruction attribute on top of the normal register swizzles called *Static (per instruction) Rounding Mode* or *Rounding Mode override*. This attribute allows statically applying a specific arithmetic rounding mode ignoring the value of RM bits in MXCSR.

Static Rounding Mode can be enabled in the encoding of the instruction by setting the *EH* bit to 1 in a register-register vector instruction. Table 2.14 shows the available rounding modes and their encoding. On top of the rounding-mode, Knights Corner also allows to set the SAE ("suppress-all-exceptions") attribute, to disable reporting any floating-point exception flag on MXCSR. This option is available, even if the instruction does not perform any kind of rounding.

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 swizzle-specified rounding mode (in the same way that they take precedence over the MXCSR.RC setting).

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}
1xx	SAE	, {sae}

Table 2.14: **Static Rounding-Mode Swizzle** available modes plus SAE.

2.4 Knights Corner Execution Environments

Knights Corner's support for 32 bit and 64 bit execution environments are similar to those found in the Intel64® Intel® Architecture Software Developer's Manual. The 64 bit execution environment of Knights Corner is shown in Figure 2.1. The layout of 512-bit vector registers and vector mask registers are shown in Figure 2.2. This section describes new features associated with the 512-bit vector registers and the 16 bit vector mask registers.

Knights Corner instructions defines two new sets of registers that hold the new vector state. The Knights Corner instructions extension uses the vector registers, the vector mask registers and/or the *x86 64* general purpose registers.

Knights Corner instructions Vector Registers. The 32 registers each store store 16 doubleword/single precision floating-point entries (or 8 quadword/double precision floating-point entries), and serve as source and destination operands for vector packed floating point and integer operations. Additionally, they may also contain memory pointer offsets used to gather and scatter data from/to memory. These registers are referenced as zmm0 through zmm31.

Vector Mask Registers. These registers specify which vector elements are operated on and written for Knights Corner instructions vector instructions. If the Nth bit of a vector mask register is set, then the Nth element of the destination vector is overridden with the result of the operation; otherwise, the element remains unchanged. A vector mask register can be set using vector compare instructions, instructions to move contents from a GP register, or a special subset of vector mask arithmetic instructions.

Knights Corner vector instructions are able to report exceptions via MXCSR flags but never cause traps as all SIMD floating-point exceptions are always masked (unlike Intel® SSE/Intel® AVX instructions in other processors, that may trap if floating-point exceptions are unmasked, depending on the value of the OM/UM/IM/PM/DM/ZM bits). The reason is that Knights Corner forces the new DUE bit (Disable Unmasked Exceptions) in the MXCSR (bit21) to be set to 1.

On Knights Corner, both single precision and double precision floating-point instructions use MXCSR.DAZ and MXCSR.FZ to decide whether to treat input denormals as zeros or to flush tiny results to zero (the latter are in most cases - but not always - denormal results which are flushed to zero when MXCSR.FZ is set to 1; see the IEEE Standard 754-2008, section 7.5, for a definition of tiny floating-point results).

Table 2.15 shows the bit layout of the MXCSR control register.

MXCSR bit 20 is reserved, however it is not reported as Reserved by MXCSR_MASK. Setting this bit will result in undefined behavior

General-purpose registers. The sixteen general-purpose registers are available in Knights Corner's 64 bit

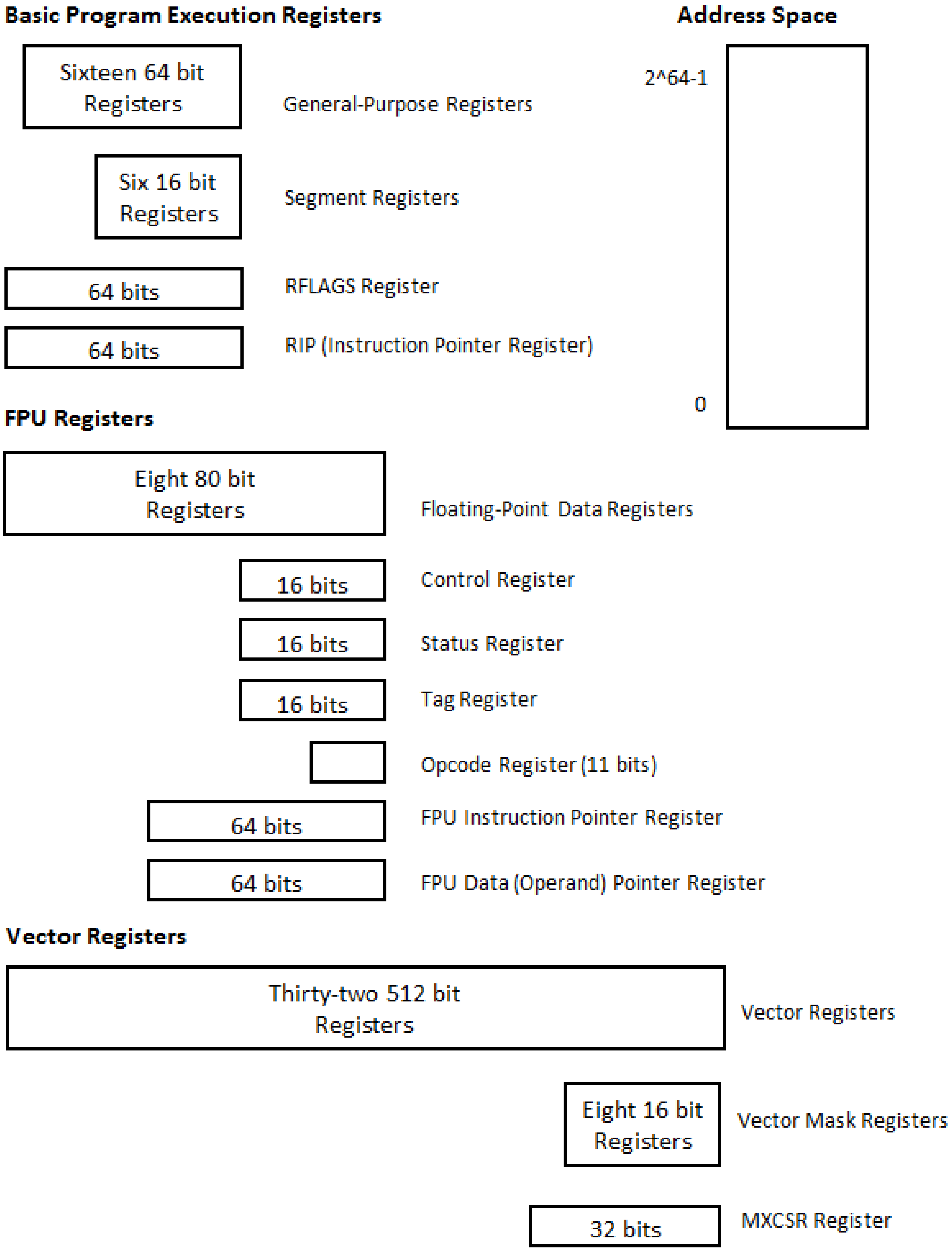


Figure 2.1: 64 bit Execution Environment

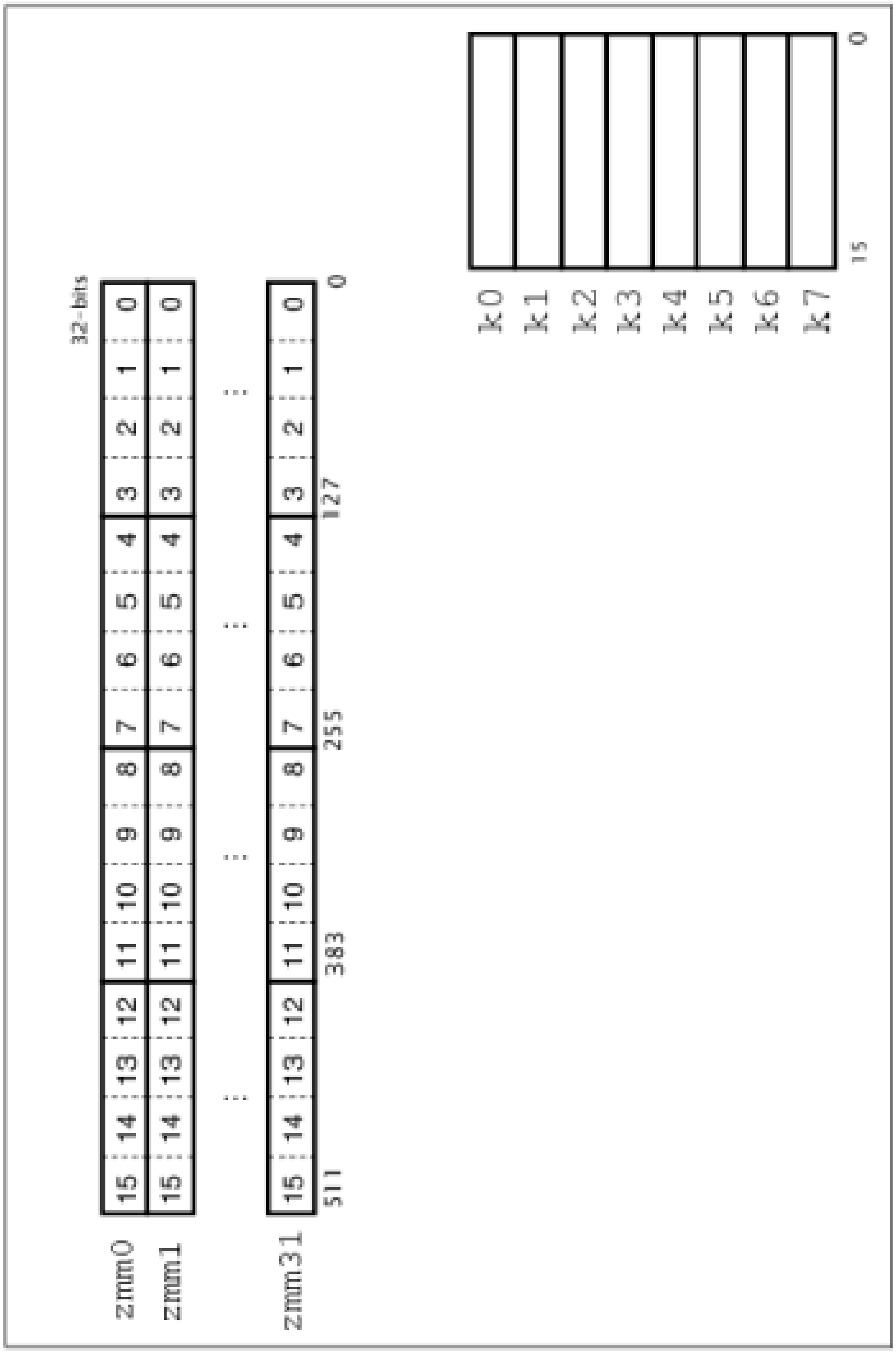


Figure 2.2: Vector and Vector Mask Registers

mode execution environment. These registers are identical to those available in the 64 bit execution environment described in the Intel64® Intel® Architecture Software Developer's Manual.

EFLAGS register. R/EFLAGS are updated by instructions according to the Intel64® Intel® Architecture Software Developer's Manual. Additionally, it is also updated by Knights Corner's KORTEST instruction.

FCW and FSW registers. Used by x87 instruction set extensions to set rounding modes, exception masks and flags in the case of the FCW, and to keep track of exceptions in the case of the FSW.

x87 stack. An eight-element stack used to perform floating-point operations on 32/64/80-bit floating-point data using the x87 instruction set.

Bit fields	Field	Description
31-22	Reserved	Reserved bits
21	DUE	Disable Unmasked Exceptions (always set to 1)
20-16	Reserved	Reserved bits
15	FZ	Flush To Zero
14-13	RC	Rounding Control
12-7	Reserved	Reserved bits (IM/DM/ZM/OM/UM/PM in other proliferations)
6	DAZ	Denormals Are Zeros
5	PE	Precision Flag
4	UE	Underflow Flag
3	OE	Overflow Flag
2	ZE	Divide-by-Zero Flag
1	DE	Denormal Operation Flag
0	IE	Invalid Operation Flag

Table 2.15: **MXCSR** bit layout. Note: MXCSR bit 20 is reserved, however it is not reported as Reserved by MXCSR_MASK. Setting this bit will result in undefined behavior

Chapter 3

Knights Corner Instruction Format

This chapter describes the instruction encoding format and assembly instruction syntax of new instructions supported by Knights Corner.

3.1 Overview

Knights Corner introduces 512-bit vector instructions operating on 512-bit vector registers (zmm0-zmm31), and offers vector mask registers (k0-k7) to support a rich set of conditional operations on data elements within the zmm registers. Vector instructions operating on zmm registers are encoded using a multi-byte prefix encoding scheme, with 62H being the 1st of the multi-byte prefix. This multi-byte prefix is referred to as MVEX in this document.

Instructions operating on the vector mask registers are encoded using another multi-byte prefix, with C4H or C5H being the 1st of the multi-byte prefix. This multi-byte prefix is similar to the VEX prefix that is defined in the "Intel® Architecture Instruction Set Architecture Programming Reference". We will refer to the C4H/C5H based VEX-like prefix as "VEX" in this document. Additionally, Knights Corner also provides a handful of new instructions operating on general-purpose registers but are encoded using VEX. In some cases, new scalar instructions supported by Knights Corner can be encoded with either MVEX or VEX.

3.2 Instruction Formats

Instructions encoded by MVEX have the format shown in Figure 3.1.

Instructions encoded by VEX have the format shown in Figure 3.2.

3.2.1 MVEX/VEX and the LOCK prefix

Any MVEX-encoded or VEX-encoded instruction with a LOCK prefix preceding the multi-byte prefix will generate an invalid opcode exception (#UD).

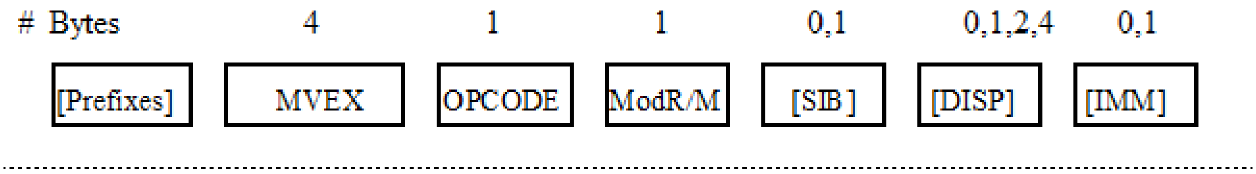


Figure 3.1: New Instruction Encoding Format with MVEX Prefix

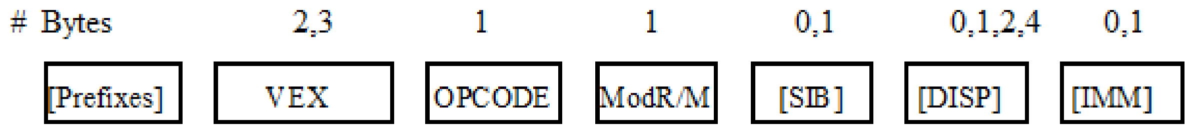


Figure 3.2: New Instruction Encoding Format with VEX Prefix

3.2.2 MVEX/VEX and the 66H, F2H, and F3H prefixes

Any MVEX-encoded or VEX-encoded instruction with a 66H, F2H, or F3H prefix preceding the multi-byte prefix will generate an invalid opcode exception (#UD).

3.2.3 MVEX/VEX and the REX prefix

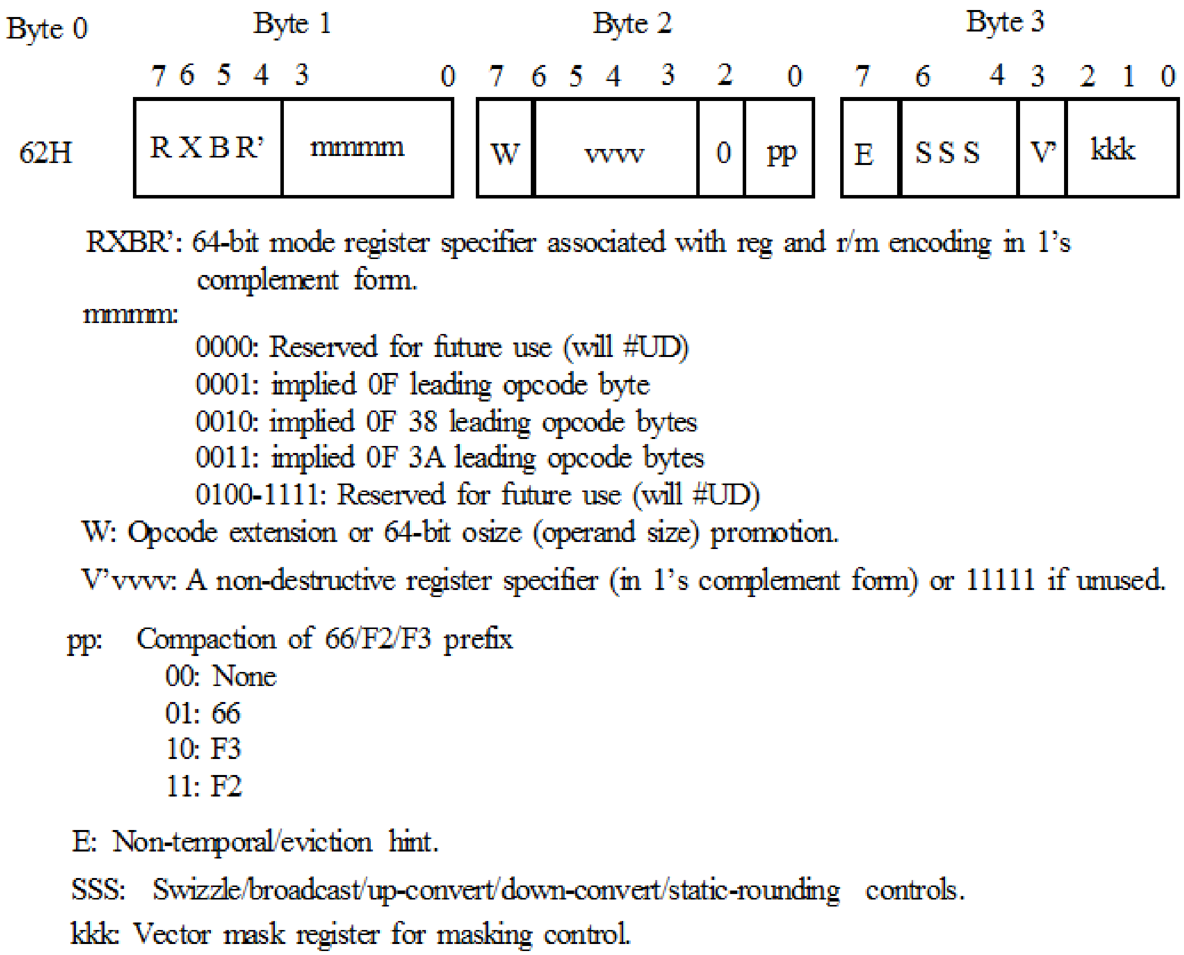
Any MVEX-encoded or VEX-encoded instruction with a REX prefix preceding the multi-byte prefix will generate an invalid opcode exception (#UD).

3.3 The MVEX Prefix

The MVEX prefix consists of four bytes that must lead with byte 62H. An MVEX-encoded instruction supports up to three operands in its syntax and is operating on vectors in vector registers or memory using a vector mask register to control the conditional processing of individual data elements in a vector. Swizzling, conversion and other operations on data elements within a vector can be encoded with bit fields in the MVEX prefix, as shown in Figure 3.3. The functionality of these bit fields is summarized below:

- 64 bit mode register specifier encoding (R, X, B, R', W, V') for memory and vector register operands (encoded in 1's complement form).
  - A vector register as source or destination operand is encoded by combining the R'R bits with the reg field, or the XB bits with the r/m field of the modR/M byte.
  - The base of a memory operand is a general purpose register encoded by combining the B bit with the r/m field. The index of a memory operand is a general purpose register encoded by combining the X bit with the SIB.index field.

  - The vector index operand in the gather/scatter instruction family is a vector register, encoded by combining the VX bits with the SIB.index field. MVEX.vvvv is not used in the gather/scatter instruction family.



RXBR': 64-bit mode register specifier associated with reg and r/m encoding in 1's complement form.

mmmm:
- 0000: Reserved for future use (will #UD)
- 0001: implied 0F leading opcode byte
- 0010: implied 0F 38 leading opcode bytes
- 0011: implied 0F 3A leading opcode bytes
- 0100-1111: Reserved for future use (will #UD)

W: Opcode extension or 64-bit osize (operand size) promotion.

V'vvvv: A non-destructive register specifier (in 1's complement form) or 11111 if unused.

pp: Compaction of 66/F2/F3 prefix
- 00: None
- 01: 66
- 10: F3
- 11: F2

E: Non-temporal/eviction hint.

SSS: Swizzle/broadcast/up-convert/down-convert/static-rounding controls.

kkk: Vector mask register for masking control.

Figure 3.3: MVEX bitfields

- Non-destructive source register specifier (applicable to the three operand syntax): This is the first source operand in the three-operand instruction syntax. It is represented by the notation, MVEX.vvvv. It can encode any of the lower 16 zmm vector registers, or using the low 3 bits to encode a vector mask register as a source operand. It can be combined with V to encode any of the 32 zmm vector registers

- Vector mask register and masking control: The MVEX.aaa field encodes a vector mask register that is used in controlling the conditional processing operation on the data elements of a 512-bit vector instruction. The MVEX.aaa field does not encode a source or a destination operand. When the encoded value of MVEX.aaa is 000b, this corresponds to "no vector mask register will act as conditional mask for the vector instruction".

- Non-temporal/eviction hint. The MVEX.E field can encode a hint to the processor on a memory referencing instruction that the data is non-temporal and can be prioritized for eviction. When an instruction encoding does not reference any memory operand, this bit may also be used to control the function of the MVEX.SSS field.

- Compaction of legacy prefixes (66H, F2H, F3H): This is encoded in the MVEX.pp field.
- Compaction of two-byte and three-byte opcode: This is encoded in the MVEX.mmmm field.
- Register swizzle/memory conversion operations (broadcast/up-convert/down-convert)/static-rounding override: This is encoded in the MVEX.SSS field.
  - Swizzle operation is supported only for register-register syntax of 512-bit vector instruction, and requires MVEX.E = 0, the encoding of MVEX.SSS determines the exact swizzle operation - see Section 2.2
  - Static rounding override only applies to register-register syntax of vector floating-point instructions, and requires MVEX.E = 1.

The MVEX prefix is required to be the last prefix and immediately precedes the opcode bytes.

3.3.1 Vector SIB (VSIB) Memory Addressing

In the gather/scatter instruction family, an SIB byte that follows the ModR/M byte can support VSIB memory addressing to an array of linear addresses. VSIB memory addressing is supported only with the MVEX prefix.

In VSIB memory addressing, the SIB byte consists of:

- The scale field (bit 7:6), which specifies the scale factor.
- The index field (bits 5:3), which is prepended with the 2-bit logical value of the MVEX.VX bits to specify the vector register number of the vector index operand; each element in the vector register specifies an index.
- The base field (bits 2:0) is prepended with the logical value of MVEX.B field to specify the register number of the base register.

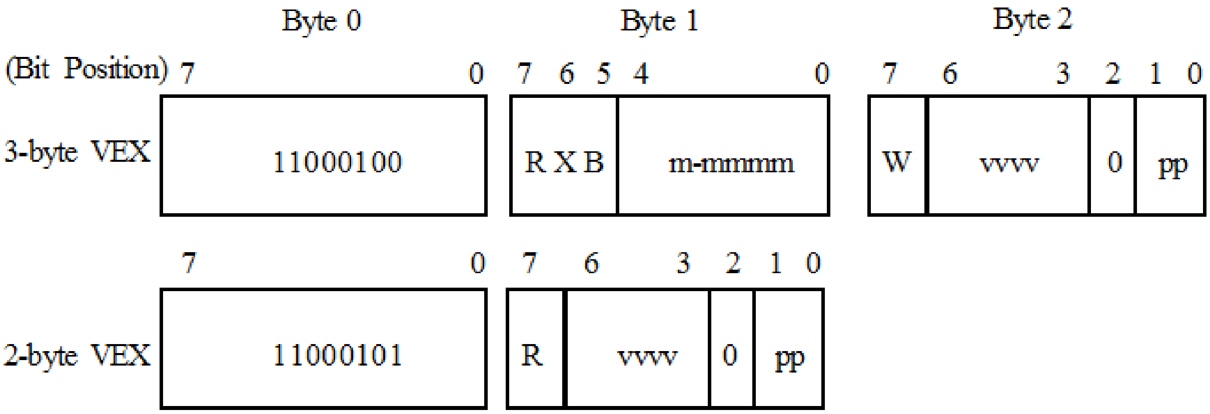
3.4 The VEX Prefix

The VEX prefix is encoded in either the two-byte form (the first byte must be C5H) or in the three-byte form (the first byte must be C4H). Beyond the first byte, the VEX prefix consists of a number of bit fields providing specific capability; they are shown in Figure 3.4.
The functionality of the bit fields is summarized below:

- 64 bit mode register specifier encoding (R, X, B, W): The R/X/B bit field is combined with the lower three bits or register operand encoding in the modR/M byte to access the upper half of the 16 registers available in 64 bit mode. The VEX.R, VEX.X, VEX.B fields replace the functionality of REX.R, REX.X, REX.B bit fields. The W bit either replaces the functionality of REX.W or serves as an opcode extension bit. The usage of the VEX.WRXB bits is explained in detail in section 2.2.1.2 of the Intel® 64 and IA-32 Architectures Software developer's manual, Volume 2A. This bit is stored in 1's complement form (bit inverted format).

- Non-destructive source register specifier (applicable to three operand syntax): this is the first source operand in the instruction syntax. It is represented by the notation, VEX.vvvv. It can encode any general-purpose register, or using only 3 bits it can encode vector mask registers. This field is encoded using 1's complement form (bit inverted form), i.e. RAX/K0 is encoded as 1111B, and R15 is encoded as 0000B.
- Compaction of legacy prefixes (66H, F2H, F3H): This is encoded in the VEX.pp field.
- Compaction of two-byte and three-byte opcode: This is encoded in the VEX.mmmmm field.

The VEX prefix is required to be the last prefix and immediately precedes the opcode bytes. It must follow any other prefixes. If the VEX prefix is present a REX prefix is not supported.



RXB: 64-bit mode register specifier associated with reg and r/m operand encoding.

m-mmmm:

00000: Reserved for future use (will #UD)
00001: implied 0F leading opcode byte
00010: implied 0F 38 leading opcode bytes
00011: implied 0F 3A leading opcode bytes
00100-11111: Reserved for future use (will #UD)

W: Opcode extension or 64-bit osize (operand size) promotion.

vvvv: A non-destructive register specifier (in 1's complement form) or 1111 if unused.

pp: Compaction of 66/F2/F3 prefix

00: None
01: 66
10: F3
11: F2

Figure 3.4: VEX bitfields

3.5 Knights Corner instructions Assembly Syntax

Knights Corner instructions supports up to three operands. The rich encoding fields for swizzle/broadcast/convert/rounding, masking control, and non-temporal hint are expressed as modifier expressions to the respective operands in the assembly syntax. A few common forms for Knights Corner assembly instruction syntax are expressed in the general form:

```
mnemonic vreg{masking modifier}, source1, transform_modifier(vreg/mem)
mnemonic vreg{masking modifier}, source1, transform_modifier(vreg/mem), imm
mnemonic mem{masking modifier}, transform_modifier(vreg)
```

The specific forms to express assembly syntax operands, modifiers, and transformations are listed in Table 3.1.

3.6 Notation

The notation used to describe the operation of each instruction is given as a sequence of control and assignment statements in C-like syntax. This document only contains the notation specifically needed for vector instructions. Standard *Intel® 64* notation may be found at *IA-32 Intel® Architecture Software Developer's Manual: Volume 2* for convenience.

When instructions are represented symbolically, the following notations are used:
label: mnemonic argument1 {write-mask}, argument2, argument3, argument4, ...

where:

- A *mnemonic* is a reserved name for a class of instruction opcodes which have the same function.
- The operands *argument1*, *argument2*, *argument3*, *argument4*, and so on are optional. There may be from one to three register operands, depending on the opcode. The leftmost operand is always the destination; for certain instructions, such as vfmadd231ps, it may be a source as well. When the second leftmost operand is a vector mask register, it may in certain cases be a destination as well, as for example with the vpsubrsetbd instruction. All other register operands are sources. There may also be additional arguments in the form of immediate operands; for example, the vcvtfxpntdq2ps instructions has a 3-bit immediate field that specifies the exponent adjustment to be performed, if any. The *write-mask* operand specifies the vector mask mask register used to control the selective updating of elements in the destination register or registers.

3.6.1 Operand Notation

In this manual we will consider vector registers from several perspectives. One perspective is is as an array of 64 bytes. Another is as an array of 16 doubleword elements. Another is an array of 8 quadword elements. Yet another is as an array of 512 bits. In the mnemonic operation description pseudo-code, registers will be addressed using bit ranges, such as:

```
i = n*32
zmm1[i+31:i]
```

This example refers to the 32 bits of the n-th doubleword element of vector register `zmm1`.

We will use a similar bit-oriented notation to describe access to vector mask registers. In the case of vector mask registers, we will usually specify a single bit, rather than a range of bits, because vector mask registers are used for predication, carry, borrow, and comparison results, and a single bit per element is enough for any of those purposes.

Using this notation, it is for example possible to test the value of the 12^{th} bit in k1 as follows:

```
if ( k1[11] == 1 ) { ... code here ... }
```

Tables 3.1 and 3.2 summarize the notation used for instruction operands and their values.

In Knights Corner instructions, the contents of vector registers are variously interpreted as floating-point values (either 32 or 64 bits), integer values, or simply doubleword values of no particular data type, depending on the instruction semantics.

3.6.2 The Displacement Bytes

Knights Corner introduces a brand new displacement representation that allows for a more compact encoding in unrolled code: compressed displacement of 8-bits, or disp8*N. Such compressed displacement is based on the assumption that the effective displacement is a multiple of the granularity of the memory access, and hence we do not need to encode the redundant low-order bits of the address offset.

Knights Corner instructions using the MVEX prefix (i.e. using encoding 62) have the following displacement options:

- No displacement
- 32 bit displacement: this displacement works exactly the same as the legacy 32 bit displacement and works at byte granularity
- Compressed 8 bit displacement (disp8*N): this displacement format substitutes the legacy 8-bit displacement in Knights Corner instructions using map 62. This displacement assumes the same granularity as the memory operand size (which is dependent on the instructions and the memory conversion function being used). Redundant low-order bits are ignored and hence, 8-bit displacements are reinterpreted so that they are multiplied by the memory operands total size in order to generate the final displacement to be used in calculating the effective address.

Note that the displacements in the MVEX vector instruction prefix are encoded in exactly the same way as regular displacements (so there are no changes in the ModRM/SIB encoding rules), with the only exception that disp8 is overloaded to disp8*N. In other words there are no changes in the encoding rules or encoding lengths, but only in the interpretation of the displacement value by hardware (which needs to scale the displacement by the size of the memory operand to obtain a byte-wise address offset).

3.6.3 Memory size and disp8*N calculation

Table 3.3 and Table 3.4 show the size of the vector (or element) being accessed in memory, which is equal to the scaling factor for compressed displacement (disp8*N). Note that some instructions work at element granularity

Notation	Meaning
zmm1	A vector register operand in the argument1 field of the instruction. The 64 byte vector registers are: zmm0 through zmm31
zmm2	A vector register operand in the argument2 field of the instruction. The 64 byte vector registers are: zmm0 through zmm31
zmm3	A vector register operand in the argument3 field of the instruction. The 64 byte vector registers are: zmm0 through zmm31
S_{f32}(zmm/m)	A vector floating-point 32 bit swizzle/conversion. Refer to Table 2.2 for register sources and Table 2.4 for memory conversions.
S_{f64}(zmm/m)	A vector floating-point 64 bit swizzle/conversion. Refer to Table 2.3 for register sources and Table 2.6 for memory conversions.
S_{i32}(zmm/m)	A vector integer 32 bit swizzle/conversion. Refer to Table 2.2 for register sources and Table 2.5 for memory conversions.
S_{i64}(zmm/m)	A vector integer 64 bit swizzle/conversion. Refer to Table 2.3 for register sources and Table 2.7 for memory conversions.
U_{f32}(m)	A floating-point 32 bit load Up-conversion. Refer to Table 2.9 for the memory conversions available for all the different datatypes.
U_{i32}(m)	An integer 32 bit load Up-conversion. Refer to Table 2.9 for the memory conversions available for all the different datatypes.
U_{f64}(m)	A floating-point 64 bit load Up-conversion. Refer to Table 2.11 for the memory conversions available for all the different datatypes.
U_{i64}(m)	An integer 64 bit load Up-conversion. Refer to Table 2.11 for the memory conversions available for all the different datatypes.
D_{f32}(zmm)	A floating-point 32 bit store Down-conversion. Refer to Table 2.12 for the memory conversions available for all the different datatypes.
D_{i32}(zmm)	An integer 32 bit store Down-conversion. Refer to Table 2.12 for the memory conversions available for all the different datatypes.
D_{f64}(zmm)	A floating-point 64 bit store Down-conversion. Refer to Table 2.13 for the memory conversions available for all the different datatypes.
D_{i64}(zmm)	An integer 64 bit store Down-conversion. Refer to Table 2.13 for the memory conversions available for all the different datatypes.
m	A memory operand.
m_t	A memory operand that may have an EH hint attribute.
mv_t	A vector memory operand that may have an EH hint attribute. This memory operand is encoded using ModRM and VSIB bytes. It can be seen as a set of pointers where each pointer is equal to $BASE + VINDEX[i] \times SCALE$
effective_address	Used to denote the full effective address when dealing with a memory operand.
imm8	An immediate byte value.
SRC[a-b]	A bit-field from an operand ranging from LSB b to MSB a.

Table 3.1: Operand Notation

Notation	Meaning
zmm1[i+31:i]	The value of the element located between bit i and bit $i + 31$ of the argument1 vector operand.
zmm2[i+31:i]	The value of the element located between bit i and bit $i + 31$ of the argument2 vector operand.
k1[i]	Specifies the i-th bit in the vector mask register k1.

Table 3.2: Vector Operand Value Notation

instead of full vector granularity at memory level, and hence should use the "element level" column in Table 3.3 and Table 3.4 (namely VLOADUNPACK, VPACKSTORE, VGATHER, and VSCATTER instructions).

Table 3.3: Size of vector or element accessed in memory for up-conversion

Function	Usage	Memory accessed / Disp8*N		
U/S$_{f32}$		No broadcast	4to16 broadcast	1to16 broadcast or element level
000	[rax] {16to16} or [rax]	64	16	4
001	[rax] {1to16}	4	NA	NA
010	[rax] {4to16}	16	NA	NA
011	[rax] {float16}	32	8	2
100	[rax] {uint8}	16	4	1
101	[rax] {sint8}	16	4	1
110	[rax] {uint16}	32	8	2
111	[rax] {sint16}	32	8	2
U/S$_{i32}$		No broadcast	4to16 broadcast	1to16 broadcast or element level
000	[rax] {16to16} or [rax]	64	16	4
001	[rax] {1to16}	4	NA	NA
010	[rax] {4to16}	16	NA	NA
011	N/A	NA	NA	NA
100	[rax] {uint8}	16	4	1
101	[rax] {sint8}	16	4	1
110	[rax] {uint16}	32	8	2
111	[rax] {sint16}	32	8	2
U/S$_{f64}$		No broadcast	4to8 broadcast	1to8 broadcast or element level
000	[rax] {8to8} or [rax]	64	32	8
001	[rax] {1to8}	8	NA	NA
010	[rax] {4to8}	32	NA	NA
011	N/A	NA	NA	NA
100	N/A	NA	NA	NA
101	N/A	NA	NA	NA
110	N/A	NA	NA	NA
111	N/A	NA	NA	NA
U/S$_{i64}$		No broadcast	4to8 broadcast	1to8 broadcast or element level
000	[rax] {8to8} or [rax]	64	32	8
001	[rax] {1to8}	8	NA	NA
010	[rax] {4to8}	32	NA	NA
011	N/A	NA	NA	NA
100	N/A	NA	NA	NA
101	N/A	NA	NA	NA
110	N/A	NA	NA	NA
111	N/A	NA	NA	NA

Table 3.4: Size of vector or element accessed in memory for down-conversion

Function	Usage	Memory accessed / Disp8*N	
D_{f32}		Regular store	Element level
000	zmm1	64	4
001	N/A	NA	NA
010	N/A	NA	NA
011	zmm1 {float16}	32	2
100	zmm1 {uint8}	16	1
101	zmm1 {sint8}	16	1
110	zmm1 {uint16}	32	2
111	zmm1 {sint16}	32	2
D_{f64}		Regular store	Element level
000	zmm1	64	8
001	N/A	NA	NA
010	N/A	NA	NA
011	N/A	NA	NA
100	N/A	NA	NA
101	N/A	NA	NA
110	N/A	NA	NA
111	N/A	NA	NA
D_{i64}		Regular store	Element level
000	zmm1	64	8
001	N/A	NA	NA
010	N/A	NA	NA
011	N/A	NA	NA
100	N/A	NA	NA
101	N/A	NA	NA
110	N/A	NA	NA
111	N/A	NA	NA

3.7 EH hint

All vector instructions that access memory provide the option of specifying a cache-line eviction hint, EH.

EH is a performance hint, and may operate in different ways or even be completely ignored in different hardware implementations. Knights Corner is designed to provide support for cache-efficient access to memory locations that have either low temporal locality of access or bursts of a few very closely bunched accesses.

There are two distinct modes of EH hint operation, one for prefetching and one for loads, stores, and load-op instructions.

The interaction of the EH hint with prefetching is summarized in Table 3.5.

EH value	Hit behavior	Miss behavior
EH not set	Make data MRU	Fetch data and make it MRU
EH set	Make data MRU	Fetch data into way #N, where N is the thread number, and make it MRU

Table 3.5: Prefetch behavior based on the **EH** (cache-line eviction hint)

The above table describes the effect of the EH bit on gather/scatter prefetches into the targeted cache (e.g. L1 for vgatherpf0dps, L2 for vgatherpf1dps). If vgatherpf0dps misses both L1 and L2, the resulting prefetch into L1 is a non-temporal prefetch into way #N of L1, but the prefetch into L2 is a normal prefetch, not a non-temporal prefetch. If you want the data to be non-temporally fetched into L2, you must use vgatherpf1dps with the EH bit set.

The operation of the EH hint with prefetching is designed to limit the cache impact of streaming data.

Note that regular prefetch instructions (like vprefetch0) do not have an embedded EH hint. Instead, the non-temporal hint is given by the opcode/mnemonic (see VPREFETCHNTA/0/1/2 descriptions for details). The same rules described in Table 3.5 still apply.

Table 3.6 summarizes the interaction of the EH hint with load and load-op instructions.

EH value	L1 hit behavior	L1 miss behavior
EH not set	Make data MRU	Fetch data and make it MRU
EH set	Make data LRU	Fetch data and make it MRU

Table 3.6: Load/load-op behavior based on the **EH** bit.

The EH bit, when used with load and load-op instructions, affects only the L1 cache behavior. Any resulting L2 misses are handled normally, regardless of the setting of the EH bit.

Table 3.7 summarizes the interaction of the EH hint with store instructions. Note that stores that write a full cache-line (no mask, no down-conversion) evict the line from L1 (invalidation) while updating the contents directly into the L2 cache. In any other case, a store with an EH hint works as a load with an EH hint.

EH value	Store type	L1 hit behavior	L1 miss behavior
EH not set		Make data MRU	Fetch data and make it MRU
EH set	No mask, no downconv.	Invalidate L1 - Update L2	Fetch data and make it MRU
EH set	Mask or downconv.	Make data LRU	Fetch data and make it MRU

Table 3.7: Store behavior based on the **EH** bit.

The EH bit, when used with load and load-op instructions, affects only the L1 cache behavior. Any resulting L2 misses are handled normally, regardless of the setting of the EH bit.

3.8 Functions and Tables Used

Some mnemonic definitions use auxiliary tables and functions to ease the process of describing the operations of the instruction. The following section describes those tables and functions that do not have an obvious meaning.

3.8.1 MemLoad and MemStore

This document uses two functions, Mem-Load and MemStore, to describe in pseudo-code memory transfers that involve no conversions or broadcasts:

- **MemLoad**: Given an address pointer, this function returns the associated data from memory. Size is defined by the explicit destination size in the pseudo-code (see for example LDMXCSR in Appendix B)
- **MemStore**: Given an address pointer, this function stores the associated data to memory. Size is defined by the explicit source data size in the pseudo-code.

3.8.2 SwizzUpConvLoad, UpConvLoad and DownConvStore

In this document, the detailed discussions of memory-accessing instructions that support datatype conversion and/or broadcast (as defined by the UpConv, SwizzUpConv, and DownConv tables in section 2.2) use the functions shown in Table 3.8 in their Operation sections (the instruction pseudo-code). These functions are used to describe any swizzle, broadcast, and/or conversion that can be performed by the instruction, as well as the actual load in the case of SwizzUpConv and UpConv. Note that zmm/m means that the source may be either a vector operand or a memory operand, depending on the ModR/M encoding.

`Swizzle/conversion used`	Function used in operation description
S_{f32}(zmm/m)	SwizzUpConvLoad$_{f32}$(zmm/m)
S_{f64}(zmm/m)	SwizzUpConvLoad$_{f64}$(zmm/m)
S_{i32}(zmm/m)	SwizzUpConvLoad$_{i32}$(zmm/m)
S_{i64}(zmm/m)	SwizzUpConvLoad$_{i64}$(zmm/m)
U_{f32}(m)	UpConvLoad$_{f32}$(m)
U_{i32}(m)	UpConvLoad$_{i32}$(m)
U_{f64}(m)	UpConvLoad$_{f64}$(m)
U_{i64}(m)	UpConvLoad$_{i64}$(m)
D_{f32}(zmm)	DownConvStore$_{f32}$(zmm) or DownConvStore$_{f32}$(zmm[xx:yy])
D_{i32}(zmm)	DownConvStore$_{i32}$(zmm) or DownConvStore$_{i32}$(zmm[xx:yy])
D_{f64}(zmm)	DownConvStore$_{f64}$(zmm) or DownConvStore$_{f64}$(zmm[xx:yy])
D_{i64}(zmm)	DownConvStore$_{i64}$(zmm) or DownConvStore$_{i64}$(zmm[xx:yy])

Table 3.8: SwizzUpConv, UpConv and DownConv function conventions

The Operation section may use *UpConvSizeOf*, which returns the final size (in bytes) of an up-converted memory element given a specified up-conversion mode. A specific subset of a memory stream may be used as a parameter for UpConv; Size of the subset is inferred by the size of destination together with the up-conversion mode.

Additionally, the Operation section may also use *DownConvStoreSizeOf*, which returns the final size (in bytes) of a downcoverted vector element given a specified down-conversion mode. A specific subset of a vector register may be used as a parameter for DownConvStore; for example, DownConvStore(zmm2[31:0]) specifies that the low 32 bits of zmm2 form the parameter for DownConv.

3.8.3 Other Functions/Identifiers

The following identifiers are used in the algorithmic descriptions:

- **Carry** - The carry bit from an addition.
- **FpMaxAbs** - The greater of the absolute values of two floating-point numbers. See the description of the VGMAXABSPS instruction for further details.
- **FpMax** - The greater of two floating-point numbers. See the description of the VGMAXPS instruction for further details.
- **FpMin** - The lesser of two floating-point numbers. See the description of the VGMINPS instruction for further details.
- **Abs** - The absolute value of a number.
- **IMax** - The greater of two signed integer numbers.
- **UMax** - The greater of two unsigned integer numbers.
- **IMin** - The lesser of two signed integer numbers.
- **UMin** - The lesser of two unsigned integer numbers.
- **CvtInt32ToFloat32** - Convert a signed 32 bit integer number to a 32 bit floating-point number.
- **CvtInt32ToFloat64** - Convert a signed 32 bit integer number to a 64 bit floating-point number.
- **CvtFloat32ToInt32** - Convert a 32 bit floating-point number to a 32 bit signed integer number using the specified rounding mode.
- **CvtFloat64ToInt32** - Convert a 64 bit floating-point number to a 32 bit signed integer number using the specified rounding mode.
- **CvtFloat32ToUint32** - Convert a 32 bit floating-point number to a 32 bit unsigned integer number using the specified rounding mode.
- **CvtFloat64ToUint32** - Convert a 64 bit floating-point number to a 32 bit unsigned integer number using the specified rounding mode.
- **CvtFloat32ToFloat64** - Convert a 32 bit floating-point number to a 64 bit floating-point number.
- **CvtFloat64ToFloat32** - Convert a 64 bit floating-point number to a 32 bit floating-point number using the specified rounding mode.
- **CvtUint32ToFloat32** - Convert an unsigned 32 bit integer number to a 32 bit floating-point number.
- **CvtUint32ToFloat64** - Convert an unsigned 32 bit integer number to a 64 bit floating-point number.
- **GetExp** - Obtains the (un-biased) exponent of a given floating-point number, returned in the form of a 32 bit floating-point number. See the description of the VGETEXPPS instruction for further details.

- **RoundToInt -** Rounds a floating-point number to the nearest integer, using the specified rounding mode. The result is a floating-point representation of the rounded integer value.
- **Borrow -** The borrow bit from a subtraction.
- **ZeroExtend -** Returns a value zero-extended to the operand-size attribute of the instruction.
- **FlushL1CacheLine -** Flushes the cache line containing the specified memory address from L1.
- **InvalidateCacheLine -** Invalidate the cache line containing the specified memory address from the whole memory cache hierarchy.
- **FetchL1CacheLine -** Prefetches the cache line containing the specified memory address into L1. See the description of the VPREFETCH1 instruction for further details.
- **FetchL2CacheLine -** Prefetches the cache line containing the specified memory address into L2. See the description of the VPREFETCH2 instruction for further details.

Chapter 4

Floating-Point Environment, Memory Addressing, and Processor State

This chapter describes the Knights Corner vector floating-point instruction exception behavior and interactions related to system programming.

4.1 Overview

Knights Corner 512-bit vector instructions that operate on floating-point data may signal exceptions related to arithmetic processing. When SIMD floating-point exceptions occur, Knights Corner supports exception reporting using exception flags in the MXCSR register, but traps (unmasked exceptions) are not supported.

Exceptions caused by memory accesses apply to vector floating-point, vector integer, and scalar instructions.

The MXCSR register (see Figure 4.1) in Knights Corner provides:

- Exception flags to indicate SIMD floating-point exceptions signaled by floating-point instructions operating on zmm registers. The flags are: IE, DE, ZE, OE, UE, PE.
- Rounding behavior and control: DAZ, FZ and RC.
- Exception Suppression: DUE (always 1)

4.1.1 Suppress All Exceptions Attribute (SAE)

Knights Corner instructions that process floating-point data support a specific feature to disable floating-point exception signaling, called SAE ("suppress all exceptions"). The SAE mode is enabled via a specific bit in the register swizzle field of the MVEX prefix (by setting the EH bit to 1). When SAE is enabled in the instruction encoding, that instruction does not report any SIMD floating-point exception in the MXCSR register. This feature is only available to the register-register format of the instructions and in combination with static rounding-mode.

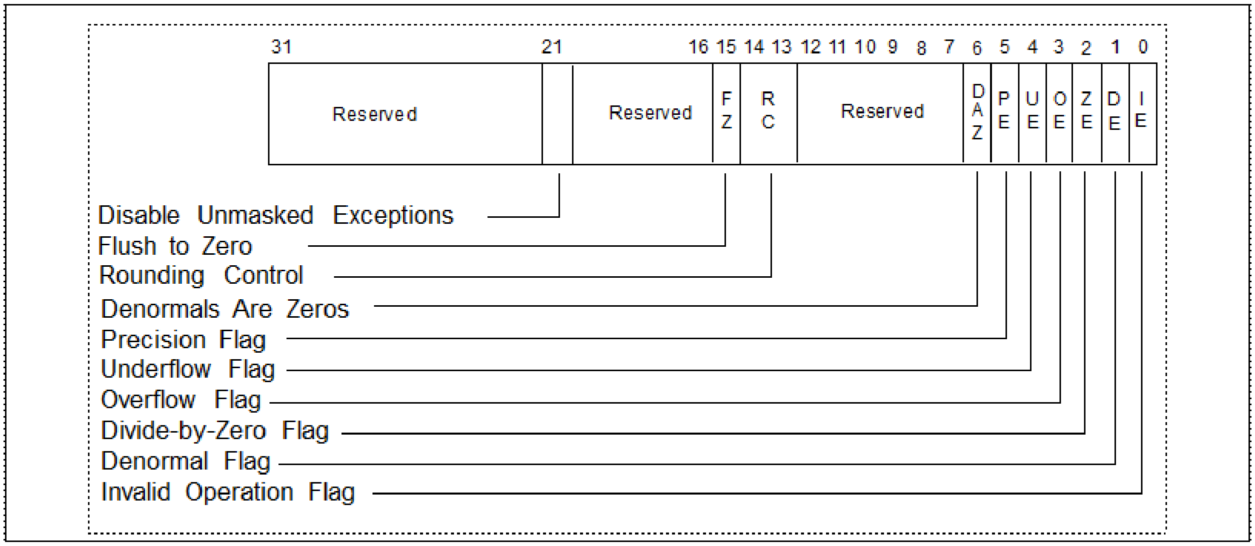


Figure 4.1: MXCSR Control/Status Register

4.1.2 SIMD Floating-Point Exceptions

SIMD floating-point exceptions are those exceptions that can be generated by Knights Corner instructions that operate on floating-point data in zmm operands. Six classes of SIMD floating-point exception flags can be signaled:

- Invalid operation (#I)
- Divide-by-zero (#Z)
- Numeric overflow (#O)
- Numeric underflow (#U)
- Inexact result (Precision) (#P)
- Denormal operand (#D)

4.1.3 SIMD Floating-Point Exception Conditions

The following sections describe the conditions that cause SIMD floating-point exceptions to be signaled, and the masked response of the processor when these conditions are detected.

When more than one exception is encountered, then the following precedence rules are applied[1].

1. Invalid-operation exception caused by sNaN operand
2. Any other invalid exception condition different from sNaN input operand

---

[1] Note that Knights Corner has no support for unmasked exceptions, so in this case the exception precedence rules have no effect. All concurrently-encountered exceptions will be reported simultaneously.

3. Denormal operand exception
4. A divide-by-zero exception
5. Overflow/underflow exception
6. Inexact result

All Knights Corner instructions floating-point exceptions are precise and are reported as soon as the instruction completes execution. The status flags from the MXCSR register set by each instruction will be the logical OR of the flags set by each of the up to 16 (or 8) individual operations. The status flags are sticky and can be cleared only via a LDMXCSR instruction.

4.1.3.1 Invalid Operation Exception (#I)

The floating-point invalid-operation exception (#I) occurs in response to an invalid arithmetic operand. The flag (IE) and mask (IM) bits for the invalid operation exception are bits 0 and 7, respectively, in the MXCSR register.

Knights Corner instructions forces all floating-point exceptions, including invalid-operation exceptions, to be masked. Thus, for the #I exception the value returned in the destination register is a QNaN, QNaN Indefinite, Integer Indefinite, or one of the source operands. When a value is returned to the destination operand, it overwrites the destination register specified by the instruction. Table 4.1 lists the invalid-arithmetic operations that the processor detects for instructions and the masked responses to these operations.

Normally, when one or more of the source operands are QNaNs (and neither is an SNaN or in an unsupported format), an invalid-operation exception is not generated. For VCMPPS and VCMPPD when the predicate is one of lt, le, nlt, or nle, a QNaN source operand does generate an invalid-operation exception.

Note that divide-by-zero exceptions (like all other floating-point exceptions) are always masked in Knights Corner.

4.1.3.2 Divide-By-Zero Exception (#Z)

The processor reports a divide-by-zero exception when a VRCP23PS instruction has a 0 operand.

Note that divide-by-zero exceptions (like all other floating-point exceptions) are always masked in Knights Corner.

4.1.3.3 Denormal Operand Exception (#D)

The processor reports a denormal operand exception when an arithmetic instruction attempts to operate on a denormal operand and the DAZ bit in the MXCSR (the "Denormals Are Zero" bit) is not set to 0 (so that denormal operands are not treated as zeros).

Note that denormal exceptions (like all other floating-point exceptions) are always masked in Knights Corner.

Condition	Masked Response
VADDNPD, VADDNPS, VADDPD, VADDPS, VADDSETSPS, VMULPD, VMULPS, VRCP23PS, VRSQRT32PS, VLOG2PS, VSCALEPS, VSUBPD, VSUBPS, VSUBRPD or an VSUBRPS instruction with an SNaN operand	Return the SNaN converted to a QNaN. For more detailed information refer to Table 4.3
VCMPPD or VCMPPS with QNaN or SNaN operand	Return 0 (except for the predicates not-equal, unordered, not-less-than, or not-less- than-or-equal, which return a 1)
VCVTPD2PS, or VCVTPS2PD instruction with an SNaN operand	Return the SNaN converted to a QNaN.
VCVTFXPNTPD2DQ, VCVTFXPNTPD2UDQ, VCVTFXPNTPS2DQ, or VCVTFXPNTPS2DQ instruction with an NaN operand	Return a 0.
VGATHERD, VMOVAPS, VLOADUNPACKHPS, VLOADUNPACKLPS, or VBROADCATSS instruction with SNaN operand and selected UpConv32 that converts from floating-point to another floating-point data type	Return the SNaN converted to a QNaN.
VPACKSTOREHPS, VPACKSTORELPS, VSCATTERDPS, or VMOVAPS instruction with SNaN operand and selected a DownConv32 that converts from float to another float datatype	Return the SNaN converted to a QNaN.
VFMADD132PD, VFMADD132PS, VFMADD213PD, VFMADD213PS, VFMADD231PD, VFMADD233PS, VFNMSUB132PD, VFNMSUB132PS, VFNMSUB213PD, VFNMSUB213PS, VFNMSUB231PD, VNMSUB231PS, VFMSUB132PD, VFMSUB132PS, VFMSUB213PD, VFMSUB213PS, VFMSUB231PD, VFMSUB231PS, VFNMADD132PD, VFNMADD132PS, VFNMADD213PD, VFNMADD213PS, VFNMADD231PD, or VFNMADD231PS instruction with an SNaN operand.	Follow rules described in Table 4.4.
VGMAXPD, VGMAXPS, VGMINPD or VGMINPS instruction with SNaN operand	Returns non NaN operand. If both operands are NaN, return first source NaN.
VGMAXABSPS instruction with SNaN operand.	Returns non NaN operand. If both operands are NaN, return first source NaN with its sign bit cleared.
Multiplication of infinity by zero	Return the QNaN floating-point Indefinite.
VGETEXPPS, VRCP23PS, VRSQRT23PS or VRNDFXPNTPS instruction with SNaN operand	Return the SNaN converted to a QNaN.
VRSQRT23PS instruction with NaN or negative value	Return the QNaN floating-point Indefinite.
Addition of opposite signed infinities or subtraction of like-signed infinities	Return the QNaN floating-point Indefinite

Table 4.1: Masked Responses of Knights Corner instructions to Invalid Arithmetic Operations

4.1.3.4 Numeric Overflow Exception (#O)

The processor reports a numeric overflow exception whenever the rounded result of an arithmetic instruction exceeds the largest allowable finite value that fits in the destination operand.

Note that overflow exceptions (like all other floating-point exceptions) are always masked in Knights Corner.

4.1.3.5 Numeric Underflow Exception (#U)

The processor signals an underflow exception whenever (a) the rounded result of an arithmetic instruction, calculated assuming unbounded exponent, is less than the smallest possible normalized finite value that will fit in the destination operand (the result is *tiny*), and (b) the final rounded result, calculated with bounded exponent determined by the destination format, is inexact.

Note that underflow exceptions (like all other floating-point exceptions) are always masked in Knights Corner.

The flush-to-zero control bit provides an additional option for handling numeric underflow exceptions in Knights Corner. If set (FZ = 1), tiny results (these are usually, but not always, denormal values) are replaced by zeros of the same sign. If not set (FZ=0) then tiny results will be rounded to 0, a denormalized value, or the smallest normalized floating-point number in the destination format, with the sign of the exact result.

4.1.3.6 Inexact Result (Precision) Exception (#P)

The inexact-result exception (also called the precision exception) occurs if the result of an operation is not exactly representable in the destination format. For example, the fraction 1/3 cannot be precisely represented in binary form. This exception occurs frequently and indicates that some (normally acceptable) accuracy has been lost. The exception is supported for applications that need to perform exact arithmetic only. In flush-to-zero mode, the inexact result exception is signaled for any tiny result. (By definition, tiny results are not zero, and are flushed to zero when MXCSR.FZ = 1 for all instructions that support this mode.)

Note that inexact exceptions (like all other floating-point exceptions) are always masked in Knights Corner.

4.2 Denormal Flushing Control

4.2.1 Denormal control in up-conversions and down-conversions

Instruction up-conversions and down-conversions follow specific denormal flushing rules, i.e. for treating input denormals as zeros and for flushing tiny results to zero:

4.2.1.1 Up-conversions

- Up-conversions from float16 to float32 ignore the MXCSR.DAZ setting and this never treat input denormals as zeros. Denormal exceptions are never signaled (the MXCSR.DE flag is never set by these operations).

- Up-conversions from any small floating-point number (namely, float16) to float32 can never generate a float32 output denormal

4.2.1.2 Down-conversions

- Down-conversions from float32 to float16 follow the MXCSR.DAZ setting to decide whether to treat input denormals as zeros or not. For input denormals, the MXCSR.DE flag is set only if MXCSR.DAZ is not set, otherwise it is left unchanged.

- Down-conversions from float32 to any integer format follow the MXCSR.DAZ setting to decide whether to treat input denormals as zeros or not (this may matter only in directed rounding modes). The MXCSR.DE status flag is never set.

- Down-conversions from float32 to any small floating-point number ignore MXCSR.FZ and always preserve output denormals.

4.3 Extended Addressing Displacements

Address displacements used by memory operands to the Knights Corner instructions vector instructions, as well as MVEX-encoded versions of VPREFETCH and CLEVICT, operate differently than do normal x86 displacements. Knights Corner instructions 8-bit displacements (i.e. when MOD.mod=01) are reinterpreted so that they are multiplied by the memory operand's total size in order to generate the final displacement to be used in calculating the effective address (32 bit displacements, which vector instructions may also use, operate normally, in the same way as for normal x86 instructions). Note that extended 8-bit displacements are still signed integer numbers and need to be sign extended.

A given vector instruction's 8-bit displacement is always multiplied by the total number of bytes of memory the instruction accesses, which can mean multiplication by 64, 32, 16, 8, 4, 2 or 1, depending on any broadcast and/or data conversion in effect. Thus when reading a 64-byte (no conversion, no broadcast) source operand, for example via

```
vmovaps zmm0, [rsi]
```

the **encoded** 8-bit displacement is first multiplied by 64 (shifted left by 6) before being used in the effective address calculation. For

```
vbroadcastss zmm0, [rsi]{uint16}   //  {1to16} broadcast of {uint16} data
```

however, the **encoded** displacement would be multiplied by 2. Note that for MVEX versions of VPREFETCH and CLEVICT, we always use disp8*64; for VEX versions we use the standard x86 disp8 displacement.

The use of disp8*N makes it possible to avoid using 32 bit displacements with vector instructions most of the time, thereby reducing code size and shrinking the required size of the paired-instruction decode window by 3 bytes. Disp8*N overcomes disp8 limitations, as it is simply too small to access enough vector operands to be useful (only 4 64-byte operands). Moreover, although disp8*N can only generate displacements that are multiples of N, that's not a significant limitation, since Knights Corner instructions memory operands must already be aligned to the total number of bytes of memory the instruction accesses in order to avoid raising a #GP fault, and that alignment is exactly what disp8*N results in, given aligned base+index addressing.

4.4 Swizzle/up-conversion exceptions

There is a set of Knights Corner instructions that do not accept all regular forms of memory up-conversion/register swizzling and raise a #UD fault for illegal combinations. The instructions are:

- VALIGND

- VCVTDQ2PD
- VCVTPS2PD
- VCVTUDQ2PD
- VEXP223PS
- VFMADD233PS
- VLOG2PS
- VPERMD
- VPERMF32X4
- VPMADD233D
- VPSHUFD
- VRCP23PS
- VRSQRT23PS

Table 4.2 summarizes which up-conversion/swizzling primitives are allowed for every one of those instructions:

Mnemonic	None	{1to16}	{4to16}	Register swizzles	Memory Conversions
VALIGND	yes	no	no	no	no
VCVTDQ2PD	yes	yes	yes	yes	no
VCVTPS2PD	yes	yes	yes	yes	no
VCVTUDQ2PD	yes	yes	yes	yes	no
VEX223PS	yes	no	no	no	no
VFMADD233PS	yes	no	yes	no	no
VLOG2PS	yes	no	no	no	no
VPERMD	yes	no	no	no	no
VPERMF32X4	yes	no	no	no	no
VPMADD233D	yes	no	yes	no	no
VPSHUFD	yes	no	no	no	no
VRCP23PS	yes	no	no	no	no
VRSQRT23PS	yes	no	no	no	no

Table 4.2: Summary of legal and illegal swizzle/conversion primitives for special instructions.

4.5 Accessing uncacheable memory

When accessing non cacheable memory, it's important to define the amount of data that is really accessed when using Knights Corner instructions (mainly when Knights Corner instructions instructions are used to access to memory mapped I/O regions). Depending on the memory region accessed, an access may cause that a mapped device behave differently.

Knights Corner instructions, when accessing to uncacheable memory access, can be categorized in four different groups:

- regular memory read operations
- vloadunpackh*/vloadunpackl*
- vgatherd*
- memory store operations

4.5.1 Memory read operations

Any Knights Corner instructions that read from memory, apart from vloadunpackh*/vloadunpackl* and vgatherd, access as many consecutive bytes as dictated by the combination of memory SwizzUpConv modifiers.

4.5.2 vloadunpackh*/vloadunpackl*

`vloadunpackh*/vloadunpackl*` instructions are exceptions to the general rule. Those two instructions will always access 64 bytes of memory. The memory region accessed is between *effective_address & (0x3F)* and *(effective_address & (0x3F)) + 63* in both cases.

4.5.3 vgatherd*

`vgatherd` instructions are able to gather to up to 16 32 bit elements. The amount of elements accessed is determined by the number of bits set in the vector mask provided as source. *Vgatherd** instruction will access up to 16 different 64-byte memory regions when gathering the elements. Note that, depending on the implementation, only one 64-byte memory access is performed for a variable number of vector elements located in that region.

Each accessed regions will be between *element_effective_address & (0x3F)* and *(element_effective_address & (0x3F)) + 63*.

4.5.4 Memory stores

All Knights Corner instructions that perform memory store operations, update those memory positions determined by the vector mask operand. Vector mask specifies which elements will be actually stored in memory. DownConv* determine the number of bytes per element that will be modified in memory.

4.6 Floating-point Notes

4.6.1 Rounding Modes

VRNDFXPNTPS and conversion instructions with float32 sources, such as VCVTFXPNTPS2DQ, support four selectable rounding modes: round to nearest (even), round toward negative infinity (round down), round toward positive infinity (round up), and round toward zero, These are the standard IEEE rounding modes; see *IA-32 Intel® Architecture Software Developer's Manual: Volume 1*, Section 4.8.4, for details.

Knights Corner introduces general support for all four rounding-modes mandated for binary floating-point arithmetic by the IEEE Standard 754-2008.

4.6.1.1 Swizzle-explicit rounding modes

Knights Corner introduces the option of specifying the rounding-mode per instruction via a specific register swizzle mode (by setting the EH bit to 1). This specific rounding-mode takes precedence over whatever MXCSR.RC specifies.

For those instructions (like VRNDFXPNTPS) where an explicit rounding-mode is specified via immediate, this immediate takes precedence over a swizzle-explicit rounding-mode embedded into the encoding of the instruction.

The priority of the rounding-modes of an instruction hence becomes (from highest to lowest):

1. Rounding mode specified in the instruction immediate (if any)
2. Rounding mode specified is the instruction swizzle attribute
3. Rounding mode specified in RC bits of the MXCSR

4.6.1.2 Definition and propagation of NaNs

The IA-32 architecture defines two classes of NaNs: quiet NaNs (QNaNs) and signaling NaNs (SNaNs). Quiet NaNs have 1 as their first fraction bit, SNaNs have 0 as their first fraction bit. An SNaN is quieted by setting its first first fraction bit to 1. The class of a NaN (quiet or signaling) is preserved when converting between different precisions.

The processor never generates an SNaN as a result of a floating-point operation with no SNaN operands, so SNaNs must be present in the input data or have to be inserted by the software.

QNaNs are allowed to propagate through most arithmetic operations without signaling an exception. Note also that Knights Corner instructions do not trap for arithmetic exceptions, as floating-point exceptions are always masked.

If any operation has one or more NaN operands then the result, in most cases, is a QNaN that is one of the input NaNs, quieted if it is an SNaN. This is chosen as the first NaN encountered when scanning the operands from left to right, as presented in the instruction descriptions from Chapter 6.

If any floating-point operation with operands that are not NaNs leads to an indefinite result (e.g. $0/0$, $0 \times \infty$, or $\infty - \infty$), the result will be QNaN Indefinite: 0xFFC00000 for 32 bit operations and 0xFFF8000000000000 for

64 bit operations.

When operating on NaNs, if the instruction does not define any other behavior, Table 4.3 describes the NaN behavior for unary and binary instructions. Table 4.4 shows the NaN behavior for ternary fused multiply and add/sub operations. This table can be derived by considering the operation as a concatenation of two binary operations. The first binary operation, the multiply, produces the product. The second operation uses the product as the first operand for the addition.

Source operands	Result
SNaN	SNaN source operand, converted into a QNaN
QNaN	QNaN source operand
SNaN and QNaN	First operand (if this operand is an SNaN, it is converted to a QNaN)
Two SNaNs	First operand converted to a QNaN
Two QNaNs	First operand
SNaN and a floating-point value	SNaN source operand, converted into a QNaN
QNaN and a floating-point value	QNaN source operand

Table 4.3: Rules for handling NaNs for unary and binary operations.

4.6.1.3 Signed Zeros

Zero can be represented as a $+0$ or a -0 depending on the sign bit. Both encodings are equal in value. The sign of a zero result depends on the operation being performed and the rounding mode being used.

Knights Corner instructions introduces the fused "multiply and add'' and "multiply and sub'' operations. These consist of a multiplication (whose sign is possibly negated) followed by an addition or subtraction, all calculated with just one rounding error.

The sign of the multiplication result is the exclusive-or of the signs of the multiplier and multiplicand, regardless of the rounding mode (a positive number has a sign bit of 0, and a negative one, a sign bit of 1).

The sign of the addition (or subtraction) result is in general that of the exact result. However, when this result is exactly zero, special rules apply: when the sum of two operands with opposite signs (or the difference of two operands with like signs) is exactly zero, the sign of that sum (or difference) is $+0$ in all rounding modes, except round down; in that case, the sign of an exact zero sum (or difference) is -0. This is true even if the operands are zeros, or denormals treated as zeros because MXCSR.DAZ is set to 1. Note that $x + x = x - (-x)$ retains the same sign as x even when x is zero; in particular, $(+0) + (+0) = +0$, and $(-0) + (-0) = -0$, in all rounding modes.

When $(a \times b) \pm c$ is exactly zero, the sign of fused multiply-add/subtract shall be determined by the rules above for a sum of operands. When the exact result of $\pm(a \times b) \pm c$ is non-zero yet the final result is zero because of rounding, the zero result takes the sign of the exact result.

The result for "fused multiply and add" follows by applying the following algorithm:

- (x_d, y_d, z_d) =DAZ applied to (Src1, Src2, Src3) (denormal operands, if any, are treated as zeros of the same sign as the operand; other operands are not changed)
- $Result_d = x_d \times y_d + z_d$ computed exactly then rounded to the destination precision.

Src1	Src2	Src3	vfmadd231ps vfmsub231ps vfnmadd231ps vmadd231pd vfnmsub231pd vfmsub231pd vfnmadd231pd	vfmadd132ps vfnmsub132ps vfmsub132ps vfnmadd132ps vmadd132pd vfnmsub132pd vfmsub132pd vfnmadd132pd	vfmadd213ps vfnmsub213ps vfmsub213ps vfnmadd213ps vmadd213pd vfnmsub213pd vfmsub213pd vfnmadd213pd	vfmadd233ps[a]
NaN$_1$,	NaN$_2$,	NaN$_3$	qNaN$_2$	qNaN$_1$	qNaN$_2$	qNaN$_2$
NaN$_1$,	NaN$_2$,	value	qNaN$_2$	qNaN$_1$	qNaN$_2$	qNaN$_2$
NaN$_1$,	value,	NaN$_3$	qNaN$_3$	qNaN$_1$	qNaN$_1$	qNaN$_3$
value,	NaN$_2$,	NaN$_3$	qNaN$_2$	qNaN$_3$	qNaN$_2$	qNaN$_2$
NaN$_1$,	value,	value	qNaN$_1$	qNaN$_1$	qNaN$_1$	qNaN$_1$
value,	NaN$_2$,	value	qNaN$_2$	qNaN$_2$	qNaN$_2$	qNaN$_2$
value,	vaule,	NaN$_3$	qNaN$_3$	qNaN$_3$	qNaN$_3$	qNaN$_3$

Table 4.4: Rules for handling NaNs for fused multiply and add/sub operations (ternary).

---

[a] The interpretation of the sources is slightly different for this instruction. Here the Src1 column and NaN$_1$ are associated with Src3[31:0]. Similarly the Src3 column and NaN$_3$ are associated with Src3[63:32].

- Result = FTZ applied to $Result_d$ (tiny results are replaced by zeros of the same sign; other results are not changed).

4.6.2 REX prefix and Knights Corner instructions interactions

The REX prefix is illegal in combination with Knights Corner instructions vector instructions, or with mask and scalar instructions allocated using VEX and MVEX prefixes.

Following the *Intel® 64* behavior, if the REX prefix is followed with any legacy prefix and not located just before the opcode escape, it will be ignored.

4.7 Knights Corner instructions State Save

Knights Corner does not include any explicit instruction to perform context save and restore of Knights Corner state. To perform a context save and restore we may use:

- Vector loads and stores for vector registers
- A combination of kmov plus scalar loads and stores for mask registers
- LDMXCSR/STMXCSR for the MXCSR state register

Note also that vector instructions raise a device-not-available (#NM) exceptions when CR0.TS is set. This allows to perform selective lazy save and restore of state.

4.8 Knights Corner instructions Processor State After Reset

Table 4.5 shows the state of the flags and other registers following power-up for Knights Corner.

Register	Knights Corner
EFLAGS	00000002H
EIP	0000FFF0H
CR0	60000010H2
CR2, CR3, CR4	00000000H
CS	Selector = F000H; Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed
SS, DS, ES, FS, GS	Selector = 0000H; Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed
EDX	000005xxH
EAX	04
EBX, ECX, ESI, EDI, EBP, ESP	00000000H
ST0 through ST7	Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged
x87 FPU Control Word	Pwr up or Reset: 0040H FINIT/FNINIT: 037FH
x87 FPU Status Word	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H
x87 FPU Tag Word	Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH
x87 FPU Data Operand and CS Seg. Selectors	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H
x87 FPU Data Operand and Inst. Pointers	Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H
MM0 through MM7	NA
XMM0 through XMM7	NA
k0 through k7	0000H
zmm0 through zmm31	0 (64 bytes)
MXCSR	0020_0000H
GDTR, IDTR	Base = 00000000H, Limit = FFFFH AR = Present, R/W
LDTR, Task Register	Selector = 0000H, Base = 00000000H Limit = FFFFH AR = Present, R/W
DR0, DR1, DR2, DR3	00000000H
DR6	FFFF0FF0H
DR7	00000400H
Time-Stamp Counter	Power up or Reset: 0H INIT: Unchanged
Perf. Counters and Event Select	Power up or Reset: 0H INIT: Unchanged
All Other MSRs	Power up or Reset: Undefined INIT: Unchanged
Data and Code Cache, TLBs	Invalid
MTRRs, Machine-Check	Not Implemented
APIC	Pwr up or Reset: Enabled INIT: Unchanged

Table 4.5: Processor State Following Power-up, Reset, or INIT.

Chapter 5

Instruction Set Reference

Knights Corner instructions that are described in this document follow the general documentation convention established in this chapter.

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

5.1.1 Instruction Format

The following is an example of the format used for each instruction description in this chapter.

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W1 50 /r	vaddnpd zmm1 k1, zmm2, S_{f64}(zmm3/m_i)	Add float64 vector zmm2 and float64 vector Sf64(zmm3/mt), negate the sum, and store the result in zmm1, under write-mask.
VEX.0F.W0 41 /r	kand k1 , k2	Perform a bitwise AND between k1 and k2, store result in k1

5.1.2 Opcode Notations for MVEX Encoded Instructions

In the Instruction Summary Table, the Opcode column presents the details of each instruction byte encoding using notations described in this section. For MVEX encoded instructions, the notations are expressed in the following form (including the modR/M byte if applicable, and the immediate byte if applicable):

```
MVEX.[NDS,NDD].[512].[66,F2,F3].0F/0F3A/0F38.[W0,W1] opcode [/r] [/ib]
```

- *MVEX:* indicates the presence of the MVEX prefix is required. The MVEX prefix consists of 4 bytes with the leading byte 62H.
  The encoding of various sub-fields of the MVEX prefix is described using the following notations:
  - *NDS,NDD:* specifies that MVEX.vvvv field is valid for the encoding of a register operand:
    - MVEX.NDS: MVEX.vvvv encodes the first source register in an instruction syntax where the content of source registers will be preserved. To encode a vector register in the range zmm16-zmm31, the MVEX.vvvv field is pre-pended with MVEX.V'.
    - MVEX.NDD: MVEX.vvvv encodes the destination register that cannot be encoded by ModR/M:reg field. To encode a vector register in the range zmm16-zmm31, the MVEX.vvvv field is pre-pended with MVEX.V'.
    - If none of NDS, NDD is present, MVEX.vvvv must be 1111b (i.e. MVEX.vvvv does not encode an operand).
  - *66,F2,F3:* The presence or absence of these value maps to the MVEX.pp field encodings. If absent, this corresponds to MVEX.pp=00B. If present, the corresponding MVEX.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 MVEX.pp may be considered as an implied 66H/F2H/F3H prefix.
  - *0F,0F3A,0F38:* The presence of these values maps to a valid encoding of the MVEX.mmmm field. Only three encoded values of MVEX.mmmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH and 0F38H.
  - *W0:* MVEX.W=0
  - *W1:* MVEX.W=1
  - 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.
- *opcode:* Instruction opcode.
- */r:* Indicates that the ModR/M byte of the instruction contains a register operand and an r/m operand.
- */vsib:* Indicates the memory addressing uses the vector SIB byte.
- *ib:* A 1-byte immediate operand to the instruction that follows the opcode, ModR/M bytes or scale/indexing bytes.

In general, the encoding of the MVEX.R, MVEX.X, MVEX.B, and MVEX.V' fields are not shown explicitly in the opcode column. The encoding scheme of MVEX.R, MVEX.X, MVEX.B, and MVEX.V' fields must follow the rules defined in Chapter 3.

5.1.3 Opcode Notations for VEX Encoded Instructions

In the Instruction Summary Table, the Opcode column presents the details of each instruction byte encoding using notations described in this section. For VEX encoded instructions, the notations are expressed in the following form (including the modR/M byte if applicable, the immediate byte if applicable):

```
VEX.[NDS,NDD].[66,F2,F3].0F/0F3A/0F38.[W0,W1] opcode [/r] [/ib]
```

- *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. Refer to Chapter 3 for more details on the VEX prefix.
  The encoding of various sub-fields of the VEX prefix is described using the following notations:
  - *NDS,NDD:* specifies that VEX.vvvv field is valid for the encoding of a register operand:
    - VEX.NDS: VEX.vvvv encodes the first source register in an instruction syntax where the content of source registers will be preserved.
    - VEX.NDD: VEX.vvvv encodes the destination register that cannot be encoded by ModR/M:reg field.
    - If none of NDS, NDD is present, VEX.vvvv must be 1111b (i.e. VEX.vvvv does not encode an operand). The VEX.vvvv field can be encoded using either the 2-byte or 3-byte form of the VEX prefix.
  - *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 of these values 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
  - 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.
- *opcode:* Instruction opcode.
- */r:* Indicates that the ModR/M byte of the instruction contains a register operand and an r/m operand.
- *ib:* A 1-byte immediate operand to the instruction that follows the opcode, ModR/M bytes or scale/indexing bytes.
- In general, the encoding of the VEX.R, VEX.X, and VEX.B fields are not shown explicitly in the opcode column. The encoding scheme of VEX.R, VEX.X, and VEX.B fields must follow the rules defined in Chapter 3.

Chapter 6

Instruction Descriptions

This Chapter defines all of the Knights Corner instructions vector instructions. Note: Some instruction descriptions refer to the SSS or $S_2S_1S_0$, which are bits 6-4 from the MVEX prefix encoding. See Table 2.14 for more details

6.1 Vector Mask Instructions

JKNZD - Jump near if mask is not zero

Opcode	Instruction	Description
VEX.NDS.128.0F.W0 85 id	jknzd k1, rel32	Jump near if mask is not zero.
VEX.NDS.128.W0 75 ib	jknzd k1, rel8	Jump near if mask is not zero.

Description

Checks the value of source mask, and if not all mask bits are set to 0, performs a jump to the target instruction specified by the destination operand. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the instruction.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). A relative offset (rel8, rel16, or rel32) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32 bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of -128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

The instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the JKNZD instruction, and then access the target with an unconditional far jump (JMP instruction) to the other segment. For example, the following conditional far jump is illegal:

JKNZD FARLABEL;

To accomplish this far jump, use the following two instructions:

JKZD BEYOND;
JMP FARLABEL;
BEYOND:

This conditional jump is converted to code fetch of one or two cache lines, regardless of jump address or cacheability.

In 64 bit mode, operand size (OSIZE) is fixed at 64 bits. JMP Short is RIP = RIP + 8-bit offset sign extended to 64 bits. JMP Near is RIP = RIP + 32 bit offset sign extended to 64 bits.

Operation

```
if (k1[15:0]!=0)
{
   tempEIP = EIP + SignExtend(DEST);

   if(OSIZE == 16)
   {
      tempEIP = tempEIP & 0000FFFFH;
   }
   if (*tempEIP is not within code segment limit*)
   {
      #GP(0);
   }
   else
   {
      EIP = tempEIP
   }
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

None.

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#GP(0)	If the memory address is in a non-canonical form.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.

JKZD - Jump near if mask is zero

Opcode	Instruction	Description
VEX.NDS.128.0F.W0 84 id	jkzd k1, rel32	Jump near if mask is zero.
VEX.NDS.128.W0 74 ib	jkzd k1, rel8	Jump near if mask is zero.

Description

Checks the value of source mask, and if all mask bits are set to 0, performs a jump to the target instruction specified by the destination operand. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the instruction.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). A relative offset (rel8, rel16, or rel32) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32 bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of -128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

The instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the JKNZD instruction, and then access the target with an unconditional far jump (JMP instruction) to the other segment. For example, the following conditional far jump is illegal:

JKZD FARLABEL;

To accomplish this far jump, use the following two instructions:

JKNZD BEYOND;
JMP FARLABEL;
BEYOND:

This conditional jump is converted to code fetch of one or two cache lines, regardless of jump address or cacheability.

In 64 bit mode, operand size (OSIZE) is fixed at 64 bits. JMP Short is RIP = RIP + 8-bit offset sign extended to 64 bits. JMP Near is RIP = RIP + 32 bit offset sign extended to 64 bits.

Operation

```
if (k1[15:0]==0)
{
   tempEIP = EIP + SignExtend(DEST);

   if(OSIZE == 16)
   {
      tempEIP = tempEIP & 0000FFFFH;
   }
   if (*tempEIP is not within code segment limit*)
   {
      #GP(0);
   }
   else
   {
      EIP = tempEIP
   }
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

None.

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#GP(0)	If the memory address is in a non-canonical form.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.

KAND - AND Vector Mask

Opcode	Instruction	Description
VEX.128.0F.W0 41 /r	kand k1, k2	Perform a bitwise AND between vector masks k1 and k2 and store the result in vector mask k1.

Description

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

Operation

```
for (n = 0; n < 16; n++) {
  k1[n] = k1[n] & k2[n]
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

__mmask16 _mm512_kand (__mmask16, __mmask16);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

KANDN - AND NOT Vector Mask

Opcode	Instruction	Description
VEX.128.0F.W0 42 /r	kandn k1, k2	Perform a bitwise AND between NOT (vector mask k1) and vector mask k2 and store the result in vector mask k1.

Description

Performs a bitwise AND between vector mask k2, and the NOT (bitwise logical negation) of vector mask k1, and writes the result into vector mask k1.

Operation

```
for (n = 0; n < 16; n++) {
  k1[n] = (~(k1[n])) & k2[n]
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

`__mmask16 _mm512_kandn (__mmask16, __mmask16);`

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

KANDNR - Reverse AND NOT Vector Mask

Opcode	Instruction	Description
VEX.128.0F.W0 43 /r	kandnr k1, k2	Perform a bitwise AND between NOT (vector mask k2) and vector mask k1 and store the result in vector mask k1.

Description

Performs a bitwise AND between the NOT (bitwise logical negation) of vector mask k2, and the vector mask k1, and writes the result into vector mask k1.

Operation

```
for (n = 0; n < 16; n++) {
  k1[n] = ~(k2[n]) & k1[n]
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

__mmask16 _mm512_kandnr (__mmask16, __mmask16);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

KCONCATH - Pack and Move High Vector Mask

Opcode	Instruction	Description
VEX.NDS.128.0F.W0 95 /r	kconcath r64, k1, k2	Concatenate vector masks k1 and k2 into the high part of register r64.

Description

Packs vector masks k1 and k2 and moves the result to the high 32 bits of destination register r64. The rest of the destination register is zeroed.

Operation

```
TMP[15:0] = k2[15:0]
TMP[31:16] = k1[15:0]
r64[31:0] = 0
r64[63:32] = TMP
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

__int64 _mm512_kconcathi_64(__mmask16, __mmask16);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If destination is a memory operand.

KCONCATL - Pack and Move Low Vector Mask

Opcode	Instruction	Description
VEX.NDS.128.0F.W0 97 /r	kconcatl r64, k1, k2	Concatenate vector masks k1 and k2 into the low part of register r64.

Description

Packs vector masks k1 and k2 and moves the result to the low 32 bits of destination register r64. The rest of the destination register is zeroed.

Operation

```
TMP[15:0] = k2[15:0]
TMP[31:16] = k1[15:0]
r64[31:0] = TMP
r64[63:32] = 0
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

__int64 _mm512_kconcatlo_64(__mmask16, __mmask16);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If destination is a memory operand.

KEXTRACT - Extract Vector Mask From Register

Opcode	Instruction	Description
VEX.128.66.0F3A.W0 3E /r ib	kextract k1, r64, imm8	Extract field from general purpose register r64 into vector mask k1 using imm8.

Description

Extract the 16-bit field selected by imm8[1:0] from general purpose register r64 and write the result into destination mask register k1.

Operation

```
index = imm8[1:0] * 16
k1[15:0] = r64[index+15:index]
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

`__mmask16    _mm512_kextract_64(__int64, const in);`

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If source is a memory operand.

KMERGE2L1H - Swap and Merge High Element Portion and Low Portion of Vector Masks

Opcode	Instruction	Description
VEX.128.0F.W0 48 /r	kmerge2l1h k1, k2	Concatenate the low half of vector mask k2 and the high half of vector mask k1 and store the result in the vector mask k1.

Description

Move high element from vector mask register k1 into low element of vector mask register k1, and insert low element of k2 into the high portion of vector mask register k1.

Operation

```
tmp = k1[15:8]
k1[15:8] = k2[7:0]
k1[7:0] = tmp
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

__mmask16 _mm512_kmerge2l1h (__mmask16, __mmask16 k2);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

KMERGE2L1L - Move Low Element Portion into High Portion of Vector Mask

Opcode	Instruction	Description
VEX.128.0F.W0 49 /r	kmerge2l1l k1, k2	Move low half of vector mask k2 into the high half of vector mask k1.

Description

Insert low element from vector mask register k2 into high element of vector mask register k1. Low element of k1 remains unchanged.

Operation

```
k1[15:8] = k2[7:0]
*k1[7:0] remains unchanged*
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

__mmask16 _mm512_kmerge2l1l (__mmask16, __mmask16 k2);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

KMOV - Move Vector Mask

Opcode	Instruction	Description
VEX.128.0F.W0 90 /r	kmov k1, k2	Move vector mask k2 and store the result in k1.
VEX.128.0F.W0 93 /r	kmov r32, k2	Move vector mask k2 to general purpose register r32.
VEX.128.0F.W0 92 /r	kmov k1, r32	Move general purpose register r32 to vector mask k1.

Description

Either the vector mask register k2 or the general purpose register r32 is read, and its contents written into destination general purpose register r32 or vector mask register k1; however, general purpose register to general purpose register copies are not supported. When the destination is a general purpose register, the 16 bit value that is copied is zero-extended to the maximum operand size in the current mode.

Operation

```
if(DEST is a general purpose register) {
  DEST[63:16] = 0
  DEST[15:0] = k2[15:0]
} else if(DEST is vector mask and SRC is a general purpose register) {
  k1[15:0] = SRC[15:0]
} else {
  k1[15:0] = k2[15:0]
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

__mmask16	_mm512_kmov (__mmask16);
__mmask16	_mm512_int2mask (int);
int	_mm512_mask2int (__mmask16);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If source/destination is a memory operand.

KNOT - Not Vector Mask

Opcode	**Instruction**	**Description**
VEX.128.0F.W0 44 /r	knot k1, k2	Perform a bitwise NOT on vector mask k2 and store the result in k1.

Description

Performs the bitwise NOT of the vector mask k2, and writes the result into vector mask k1.

Operation

```
for (n = 0; n < 16; n++) {
  k1[n] =  ~ k2[n]
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

`__mmask16    _mm512_knot(__mmask16);`

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

KOR - OR Vector Masks

Opcode	Instruction	Description
VEX.128.0F.W0 45 /r	kor k1, k2	vector masks k1 and k2 and store the result in vector mask k1.

Description

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

Operation

```
for (n = 0; n < 16; n++) {
  k1[n] = k1[n] | k2[n]
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

```
__mmask16    _mm512_kor(__mmask16, __mmask16);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

KORTEST - OR Vector Mask And Set EFLAGS

Opcode	Instruction	Description
VEX.128.0F.W0 98 /r	kortest k1, k2	vector masks k1 and k2 and update ZF and CF EFLAGS accordingly.

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

```
CF = 1
ZF = 1
for (n = 0; n < 16; n++) {
  tmp = (k1[n] | k2[n])
  ZF &= (tmp == 0x0)
  CF &= (tmp == 0x1)
}
```

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.

Intel® C/C++ Compiler Intrinsic Equivalent

```
int   _mm512_kortestz (__mmask16, __mmask16);
int   _mm512_kortestc (__mmask16, __mmask16);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

KXNOR - XNOR Vector Masks

Opcode	Instruction	Description
VEX.128.0F.W0 46 /r	kxnor k1, k2	vector masks k1 and k2 and store the result in vector mask k1.

Description

Performs a bitwise XNOR between the vector mask k1 and the vector mask k2, and the result is written into vector mask k1.

The primary purpose of this instruction is to provide a way to set a vector mask register to 0xFFFF in a single clock; this is accomplished by selecting the source and destination to be the same mask register. In this case the result will be 0xFFFF regardless of the original contents of the register.

Operation

```
for (n = 0; n < 16; n++) {
  k1[n] = ~(k1[n] ^ k2[n])
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

__mmask16 _mm512_kxnor (__mmask16, __mmask16);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

KXOR - XOR Vector Masks

Opcode	**Instruction**	**Description**
VEX.128.0F.W0 47 /r	kxor k1, k2	vector masks k1 and k2 and store the result in vector mask k1.

Description

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

Operation

```
for (n = 0; n < 16; n++) {
  k1[n] = k1[n] ^ k2[n]
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

__mmask16 _mm512_kxor (__mmask16, __mmask16);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

6.2 Vector Instructions

VADDNPD - Add and Negate Float64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W1 50 /r	vaddnpd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Add float64 vector zmm2 and float64 vector S_{f64}(zmm3/m_t), negate the sum, and store the result in zmm1, under write-mask.

Description

Performs an element-by-element addition between float64 vector zmm2 and the float64 vector result of the swizzle/broadcast/conversion process on memory or float64 vector zmm3, then negates the result. The final result is written into float64 vector zmm1.

Note that all the operations must be performed before rounding.

x	y	RN/RU/RZ			RD		
+0	+0	(-0)	+ (-0)	= -0	(-0)	+ (-0)	= -0
+0	-0	(-0)	+ (+0)	= +0	(-0)	+ (+0)	= -0
-0	+0	(+0)	+ (-0)	= +0	(+0)	+ (-0)	= -0
-0	-0	(+0)	+ (+0)	= +0	(+0)	+ (+0)	= +0

Table 6.1: VADDN outcome when adding zeros depending on rounding-mode. See Signed Zeros in Section 4.6.1.3 for other cases with a result of zero.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = (-zmm2[i+63:i]) + (-tmpSrc3[i+63:i])
  }
```

```
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
(MXCSR.FZ)?  YES : NO
```

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_addn_pd(__m512d, __m512d);
__m512d    _mm512_mask_addn_pd(__m512d, __mmask8, __m512d, __m512d);
```

Memory Up-conversion: S_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S_{f64}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VADDNPS - Add and Negate Float32 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W0 50 /r	vaddnps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Add float32 vector zmm2 and float32 vector S_{f32}(zmm3/m_t), negate the sum, and store the result in zmm1, under write-mask.

Description

Performs an element-by-element addition between float32 vector zmm2 and the float32 vector result of the swizzle/broadcast/conversion process on memory or float32 vector zmm3, then negates the result. The final result is written into float32 vector zmm1.

Note that all the operations must be performed before rounding.

x	y	RN/RU/RZ			RD		
+0	+0	(-0)	+ (-0)	= -0	(-0)	+ (-0)	= -0
+0	-0	(-0)	+ (+0)	= +0	(-0)	+ (+0)	= -0
-0	+0	(+0)	+ (-0)	= +0	(+0)	+ (-0)	= -0
-0	-0	(+0)	+ (+0)	= +0	(+0)	+ (+0)	= +0

Table 6.2: VADDN outcome when adding zeros depending on rounding-mode. See Signed Zeros in Section 4.6.1.3 for other cases with a result of zero.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = (-zmm2[i+31:i]) + (-tmpSrc3[i+31:i])
  }
```

```
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
    (MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :

```
    (MXCSR.FZ)? YES : NO
```

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_addn_ps (__m512, __m512);
__m512    _mm512_mask_addn_ps (__m512, __mmask16, __m512, __m512);
```

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VADDPD - Add Float64 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F.W1 58 /r	vaddpd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Add float64 vector zmm2 and float64 vector S_{f64}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element addition between float64 vector zmm2 and the float64 vector result of the swizzle/broadcast/conversion process on memory or float64 vector zmm3. The result is written into float64 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = zmm2[i+63:i] + tmpSrc3[i+63:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

```
Treat Input Denormals As Zeros :
        (MXCSR.DAZ)?  YES : NO

Flush Tiny Results To Zero :
        (MXCSR.FZ)?  YES : NO
```

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d   _mm512_add_pd(__m512d, __m512d);
__m512d   _mm512_mask_add_pd(__m512d, __mmask8, __m512d , __m512d);
```

Memory Up-conversion: S_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VADDPS - Add Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.0F.W0 58 /r	vaddps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Add float32 vector zmm2 and float32 vector S_{f32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element addition between float32 vector zmm2 and the float32 vector result of the swizzle/broadcast/conversion process on memory or float32 vector zmm3. The result is written into float32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = zmm2[i+31:i] + tmpSrc3[i+31:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
```
(MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :
```
(MXCSR.FZ)? YES : NO
```

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_add_ps (__m512, __m512);
__m512    _mm512_mask_add_ps (__m512, __mmask16, __m512, __m512);
```

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VADDSETSPS - Add Float32 Vectors and Set Mask to Sign

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 CC /r	vaddsetsps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Add float32 vector zmm2 and float32 vector S_{f32}(zmm3/m_t) and store the sum in zmm1 and the sign from the sum in k1, under write-mask.

Description

Performs an element-by-element addition between float32 vector zmm2 and the float32 vector result of the swizzle/broadcast/conversion process on memory or float32 vector zmm3. The result is written into float32 vector zmm1.

In addition, the sign of the result for the n-th element is written into the n-th bit of vector mask k1.

It is the sign bit of the final result that gets copied to the destination, as opposed to the result of comparison with zero.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1 and k1. Elements in zmm1 and k1 with the corresponding bit clear in k1 register retain their previous value.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = zmm2[i+31:i] + tmpSrc3[i+31:i]
    k1[n] = zmm1[i+31]
  }
```

```
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
```
    (MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :
```
    (MXCSR.FZ)? YES : NO
```

Intel® C/C++ Compiler Intrinsic Equivalent

__m512 _mm512_addsets_ps (__m512, __m512, __mmask16*);
__m512 _mm512_mask_addsets_ps (__m512, __mmask16, __m512 , __m512, __mmask16*);

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S_{f32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_addsets_ps (__m512, __m512, __mmask16*);
__m512    _mm512_mask_addsets_ps (__m512, __mmask16, __m512 , __m512,
          __mmask16*);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is

	in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If no write mask is provided or selected write-mask is k0.

VALIGND - Align Doubleword Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F3A.W0 03 /r ib	valignd zmm1 {k1}, zmm2, zmm3/m_t, offset	Shift right and merge vectors zmm2 and zmm3/m_t with doubleword granularity using offset as number of elements to shift, and store the final result in zmm1, under write-mask.

Description

Concatenates and shifts right doubleword elements from vector zmm2 and memory/vector zmm3. The result is written into vector zmm1.

No swizzle, broadcast, or conversion is performed by this instruction.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
src[511:0] = zmm3/$m_t$

// Concatenate sources
tmp[511:0] = src[511:0]
tmp[1023:512] = zmm2[511:0]

// Shift right doubleword elements
SHIFT = imm8[3:0]
tmp[1023:0] = tmp[1023:0] >> (32*SHIFT)

// Apply write-mask
for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] = tmp[i+31:i]
  }
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_alignr_epi32 (__m512i, __m512i, const int);
__m512i    _mm512_mask_alignr_epi32 (__m512i, __mmask16, __m512i, __m512i, const int);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.
	This instruction does not support any SwizzUpConv different from the default value (no broadcast, no conversion). If SwizzUpConv function is set to any value different than "no action", then an Invalid Opcode fault is raised. This includes register swizzles.

VBLENDMPD - Blend Float64 Vectors using the Instruction Mask

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W1 65 /r	vblendmpd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Blend float64 vector zmm2 and float64 vector S_{f64}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element blending between float64 vector zmm2 and the float64 vector result of the swizzle/broadcast/conversion process on memory or float64 vector zmm3, using the instruction mask as selector. The result is written into float64 vector zmm1.

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

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = tmpSrc3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(tmpSrc3/$m_t$)
}

for (n = 0; n < 8; n++) {
   if(k1[n]==1 or *no write-mask*) {
      zmm1[i+63:i] = tmpSrc3[i+63:i]
   } else {
      zmm1[i+63:i] = zmm2[i+63:i]
   }
}
```

SIMD Floating-Point Exceptions

None.

Denormal Handling

Treat Input Denormals As Zeros :
`NO`

Flush Tiny Results To Zero :
`NO`

Memory Up-conversion: S_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S_{f64}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m5128    _mm512_mask_blend_pd (__mmask8, __m5128, __m5128);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VBLENDMPS - Blend Float32 Vectors using the Instruction Mask

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 65 /r	vblendmps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Blend float32 vector zmm2 and float32 vector S_{f32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element blending between float32 vector zmm2 and the float32 vector result of the swizzle/broadcast/conversion process on memory or float32 vector zmm3, using the instruction mask as selector. The result is written into float32 vector zmm1.

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

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = tmpSrc3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(tmpSrc3/$m_t$)
}

for (n = 0; n < 16; n++) {
   if(k1[n]==1 or *no write-mask*) {
      zmm1[i+31:i] = tmpSrc3[i+31:i]
   } else {
      zmm1[i+31:i] = zmm2[i+31:i]
   }
}
```

SIMD Floating-Point Exceptions

Invalid.

Denormal Handling

Treat Input Denormals As Zeros :
`NO`

Flush Tiny Results To Zero :
`NO`

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_mask_blend_ps (__mmask16, __m512, __m512);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VBROADCASTF32X4 - Broadcast 4xFloat32 Vector

Opcode	Instruction	Description
MVEX.512.66.0F38.W0 1A /r	vbroadcastf32x4 zmm1 {k1}, $U_{f32}(m_t)$	Broadcast 4xfloat32 vector $U_{f32}(m_t)$ into vector zmm1, under write-mask.

Description

The 4, 8 or 16 bytes (depending on the conversion and broadcast in effect) at memory address m_t are broadcast and/or converted to a float32 vector. The result is written into float32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
// {4to16}
tmpSrc2[127:0] = UpConvLoad$_{f32}$($m_t$)
for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    j = i & 0x7F
    zmm1[i+31:i] = tmpSrc2[j+31:j])
  }
}
```

Flags Affected

Invalid.

Memory Up-conversion: U$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	16
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float16 to float32	[rax] {float16}	8
100	uint8 to float32	[rax] {uint8}	4
101	sint8 to float32	[rax] {sint8}	4
110	uint16 to float32	[rax] {uint16}	8
111	sint16 to float32	[rax] {sint16}	8

Intel® C/C++ Compiler Intrinsic Equivalent

__m512 _mm512_extload_ps (void const*,_MM_UPCONV_PS_ENUM, _MM_BROADCAST32_ENUM, int);
__m512 _mm512_mask_extload_ps (__m512, __mmask16, void const*,_MM_UPCONV_PS_ENUM, _MM_BROADCAST32_ENUM, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VBROADCASTF64X4 - Broadcast 4xFloat64 Vector

Opcode	Instruction	Description
MVEX.512.66.0F38.W1 1B /r	vbroadcastf64x4 zmm1 {k1}, $U_{f64}(m_t)$	Broadcast 4xfloat64 vector $U_{f64}(m_t)$ into vector zmm1, under write-mask.

Description

The 32 bytes at memory address m_t are broadcast to a float64 vector. The result is written into float64 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
// {4to8}
tmpSrc2[255:0] = UpConvLoad_{f64}($m_t$)
for (n = 0; n < 8; n++) {
  if (k1[n] != 0) {
    i = 64*n
    j = i & 0xFF
    zmm1[i+63:i] = tmpSrc2[j+63:j])
  }
}
```

Flags Affected

None.

Memory Up-conversion: U$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	32
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

__m512d	_mm512_extload_pd (void const*,_MM_UPCONV_PD_ENUM, _MM_BROADCAST64_ENUM, int);
__m512d	_mm512_mask_extload_pd (__m512, __mmask8, void const*, _MM_UPCONV_PD_ENUM, _MM_BROADCAST64_ENUM, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VBROADCASTI32X4 - Broadcast 4xInt32 Vector

Opcode	Instruction	Description
MVEX.512.66.0F38.W0 5A /r	vbroadcasti32x4 zmm1 {k1}, $U_{i32}(m_t)$	Broadcast 4xint32 vector $U_{i32}(m_t)$ into vector zmm1, under write-mask.

Description

The 4, 8 or 16 bytes (depending on the conversion and broadcast in effect) at memory address m_t are broadcast and/or converted to a int32 vector. The result is written into int32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
// {4to16}
tmpSrc2[127:0] = UpConvLoad_i32($m_t$)
for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    j = i & 0x7F
    zmm1[i+31:i] = tmpSrc2[j+31:j])
  }
}
```

Flags Affected

None.

Memory Up-conversion: U_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	16
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	4
101	sint8 to sint32	[rax] {sint8}	4
110	uint16 to uint32	[rax] {uint16}	8
111	sint16 to sint32	[rax] {sint16}	8

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i _mm512_extload_epi32 (void const*,_MM_UPCONV_EPI32_ENUM, _MM_BROADCAST32_ENUM, int);
__m512i _mm512_mask_extload_epi32 (__m512i, __mmask16, void const*, _MM_UPCONV_EPI32_ENUM, _MM_BROADCAST32_ENUM, int);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VBROADCASTI64X4 - Broadcast 4xInt64 Vector

Opcode	**Instruction**	**Description**
MVEX.512.66.0F38.W1 5B /r	vbroadcasti64x4 zmm1 {k1}, $U_{i64}(m_t)$	Broadcast 4xint64 vector $U_{i64}(m_t)$ into vector zmm1, under write-mask.

Description

The 32 bytes at memory address m_t are broadcast to a int64 vector. The result is written into int64 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
// {4to8}
tmpSrc2[255:0] = UpConvLoad$_{i64}$($m_t$)
for (n = 0; n < 8; n++) {
  if (k1[n] != 0) {
    i = 64*n
    j = i & 0xFF
    zmm1[i+63:i] = tmpSrc2[j+63:j])
  }
}
```

Flags Affected

None.

Memory Up-conversion: U$_{i64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	32
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i  _mm512_extload_epi64 (void const*,_MM_UPCONV_EPI64_ENUM,
         _MM_BROADCAST64_ENUM, int);
__m512i  _mm512_mask_extload_epi64 (__m512i, __mmask8, void const*,
         _MM_UPCONV_EPI64_ENUM, _MM_BROADCAST64_ENUM, int);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VBROADCASTSD - Broadcast Float64 Vector

Opcode	Instruction	Description
MVEX.512.66.0F38.W1 19 /r	vbroadcastsd zmm1 {k1}, $U_{f64}(m_t)$	Broadcast float64 vector $U_{f64}(m_t)$ into vector zmm1, under write-mask.

Description

The 8 bytes at memory address m_t are broadcast to a float64 vector. The result is written into float64 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
// {1to8}
tmpSrc2[63:0] = UpConvLoad$_{f64}$($m_t$)
for (n = 0; n < 8; n++) {
  if (k1[n] != 0) {
    i = 64*n
    zmm1[i+63:i] = tmpSrc2[63:0]
  }
}
```

Flags Affected

None.

Memory Up-conversion: U_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d  _mm512_extload_pd (void const*,_MM_UPCONV_PD_ENUM, _MM_BROADCAST64_ENUM, int);
__m512d  _mm512_mask_extload_pd (__m512, __mmask8, void const*, _MM_UPCONV_PD_ENUM, _MM_BROADCAST64_ENUM, int);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VBROADCASTSS - Broadcast Float32 Vector

Opcode	Instruction	Description
MVEX.512.66.0F38.W0 18 /r	vbroadcastss zmm1 {k1}, $U_{f32}(m_t)$	Broadcast float32 vector $U_{f32}(m_t)$ into vector zmm1, under write-mask.

Description

The 1, 2, or 4 bytes (depending on the conversion and broadcast in effect) at memory address m_t are broadcast and/or converted to a float32 vector. The result is written into float32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
// {1to16}
tmpSrc2[31:0] = UpConvLoad$_{f32}$($m_t$)
for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] = tmpSrc2[31:0]
  }
}
```

Flags Affected

Invalid.

Memory Up-conversion: U$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float16 to float32	[rax] {float16}	2
100	uint8 to float32	[rax] {uint8}	1
101	sint8 to float32	[rax] {sint8}	1
110	uint16 to float32	[rax] {uint16}	2
111	sint16 to float32	[rax] {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

__m512 _mm512_extload_ps (void const*,_MM_UPCONV_PS_ENUM, _MM_BROADCAST32_ENUM, int);
__m512 _mm512_mask_extload_ps (__m512, __mmask16, void const*, _MM_UPCONV_PS_ENUM, _MM_BROADCAST32_ENUM, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VCMPPD - Compare Float64 Vectors and Set Vector Mask

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F.W1 C2 /r ib	vcmppd k2 {k1}, zmm1, S_{f64}(zmm2/m_t), imm8	Compare between float64 vector zmm1 and float64 vector S_{f64}(zmm2/m_t) and store the result in k2, under write-mask.

Description

Performs an element-by-element comparison between float64 vector zmm1 and the float64 vector result of the swizzle/broadcast/conversion from memory or float64 vector zmm2. The result is written into vector mask k2.

Note: If DAZ=1, denormals are treated as zeros in the comparison (original source registers untouched). untouched). $+0$ equals -0. Comparison with NaN returns false.

Infinity of like signs, are considered equals. Infinity values of either signs are considered ordered values.

Table 6.3 summarizes VCMPPD behavior, in particular showing how various NaN results can be produced.

Predicate	Imm8 enc	Description	Emulation	If NaN	QNaN operand signals invalid
{eq}	000	A $=$ B		False	No
{lt}	001	A $<$ B		False	Yes
{le}	010	A $<=$ B		False	Yes
{gt}		A $>$ B	Swap operands, use LT	False	Yes
{ge}		A $>=$ B	Swap operands, use LE	False	Yes
{unord}	011	Unordered		True	No
{neq}	100	NOT(A $=$ B)		True	No
{nlt}	101	NOT(A $<$ B)		True	Yes
{nle}	110	NOT(A $<=$ B)		True	Yes
{ngt}		NOT(A $>$ B)	Swap operands, use NLT	True	Yes
{nge}		NOT(A $>=$ B)	Swap operands, use NLE	True	Yes
{ord}	111	Ordered		False	No

Table 6.3: VCMPPD behavior

The write-mask does not perform the normal write-masking function for this instruction. While it does enable/disable comparisons, it does not block updating of the destination; instead, if a write-mask bit is 0, the corresponding destination bit is set to 0. Nonetheless, the operation is similar enough so that it makes sense to use the usual write-mask notation. This mode of operation is desirable because the result will be used directly as a write-mask, rather than the normal case where the result is used with a separate write-mask that keeps the masked elements inactive.

Immediate Format

	Comparison Type	I_2	I_1	I_0
eq	Equal	0	0	0
lt	Less than	0	0	1
le	Less than or Equal	0	1	0
unord	Unordered	0	1	1
neq	Not Equal	1	0	0
nlt	Not Less than	1	0	1
nle	Not Less than or Equal	1	1	0
ord	Ordered	1	1	1

Operation

```
switch (IMM8[2:0]) {
    case 0: OP ← EQ; break;
    case 1: OP ← LT; break;
    case 2: OP ← LE; break;
    case 3: OP ← UNORD; break;
    case 4: OP ← NEQ; break;
    case 5: OP ← NLT; break;
    case 6: OP ← NLE; break;
    case 7: OP ← ORD; break;
}

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{f64}$(zmm2/$m_t$)
}

for (n = 0; n < 8; n++) {
  k2[n] = 0
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    k2[n] = (zmm1[i+63:i] OP tmpSrc2[i+63:i]) ? 1 : 0
  }
}

k2[15:8] = 0
```

Instruction Pseudo-ops

Compilers and assemblers may implement the following pseudo-ops in addition to the standard instruction op:

Pseudo-Op	Implementation
vcmpeqpd k2 {k1}, zmm1, S_d(zmm2/m_t)	vcmppd k2 {k1}, zmm1, S_d(zmm2/m_t), {eq}
vcmpltpd k2 {k1}, zmm1, S_d(zmm2/m_t)	vcmppd k2 {k1}, zmm1, S_d(zmm2/m_t), {lt}
vcmplepd k2 {k1}, zmm1, S_d(zmm2/m_t)	vcmppd k2 {k1}, zmm1, S_d(zmm2/m_t), {le}
vcmpunordpd k2 {k1}, zmm1, S_d(zmm2/m_t)	vcmppd k2 {k1}, zmm1, S_d(zmm2/m_t), {unord}
vcmpneqpd k2 {k1}, zmm1, S_d(zmm2/m_t)	vcmppd k2 {k1}, zmm1, S_d(zmm2/m_t), {neq}
vcmpnltpd k2 {k1}, zmm1, S_d(zmm2/m_t)	vcmppd k2 {k1}, zmm1, S_d(zmm2/m_t), {nlt}
vcmpnlepd k2 {k1}, zmm1, S_d(zmm2/m_t)	vcmppd k2 {k1}, zmm1, S_d(zmm2/m_t), {nle}
vcmpordpd k2 {k1}, zmm1, S_d(zmm2/m_t)	vcmppd k2 {k1}, zmm1, S_d(zmm2/m_t), {ord}

SIMD Floating-Point Exceptions

Invalid, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
Not Applicable
```

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__mmask8    _mm512_cmpeq_pd_mask (__m512d, __m512d);
__mmask8    _mm512_mask_cmpeq_pd_mask(__mmask8, __m512d, __m512d);
__mmask8    _mm51_cmplt_pd_mask(__m512d, __m512d);
__mmask8    _mm512_mask_cmplt_pd_mask(__mmask8, __m512d, __m512d);
__mmask8    _mm512_cmple_pd_mask(__m512d, __m512d);
__mmask8    _mm512_mask_cmple_pd_mask(__mmask8, __m512d, __m512d);
__mmask8    _mm512_cmpunord_pd_mask(__m512d, __m512d);
__mmask8    _mm512_mask_cmpunord_pd_mask(__mmask8, __m512d, __m512d);
__mmask8    _mm512_cmpneq_pd_mask(__m512d, __m512d);
__mmask8    _mm512_mask_cmpneq_pd_mask(__mmask8, __m512d, __m512d);
__mmask8    _mm512_cmpnlt_pd_mask(__m512d, __m512d);
__mmask8    _mm512_mask_cmpnlt_pd_mask(__mmask8, __m512d, __m512d);
__mmask8    _mm512_cmpnle_pd_mask(__m512d, __m512d);
__mmask8    _mm512_mask_cmpnle_pd_mask(__mmask8, __m512d, __m512d);
__mmask8    _mm512_cmpord_pd_mask(__m512d, __m512d);
__mmask8    _mm512_mask_cmpord_pd_mask(__mmask8, __m512d, __m512d);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VCMPPS - Compare Float32 Vectors and Set Vector Mask

Opcode	**Instruction**	**Description**
MVEX.NDS.512.0F.W0 C2 /r ib	vcmpps k2 {k1}, zmm1, S_{f32}(zmm2/m_t), imm8	Compare between float32 vector zmm1 and float32 vector S_{f32}(zmm2/m_t) and store the result in k2, under write-mask.

Description

Performs an element-by-element comparison between float32 vector zmm1 and the float32 vector result of the swizzle/broadcast/conversion from memory or float32 vector zmm2. The result is written into vector mask k2.

Note: If DAZ=1, denormals are treated as zeros in the comparison (original source registers untouched). untouched). $+0$ equals -0. Comparison with NaN returns false.

Infinity of like signs, are considered equals. Infinity values of either signs are considered ordered values.

Table 6.4 summarizes VCMPPS behavior, in particular showing how various NaN results can be produced.

Predicate	Imm8 enc	Description	Emulation	If NaN	QNaN operand signals invalid
{eq}	000	A $=$ B		False	No
{lt}	001	A $<$ B		False	Yes
{le}	010	A $<=$ B		False	Yes
{gt}		A $>$ B	Swap operands, use LT	False	Yes
{ge}		A $>=$ B	Swap operands, use LE	False	Yes
{unord}	011	Unordered		True	No
{neq}	100	NOT(A $=$ B)		True	No
{nlt}	101	NOT(A $<$ B)		True	Yes
{nle}	110	NOT(A $<=$ B)		True	Yes
{ngt}		NOT(A $>$ B)	Swap operands, use NLT	True	Yes
{nge}		NOT(A $>=$ B)	Swap operands, use NLE	True	Yes
{ord}	111	Ordered		False	No

Table 6.4: VCMPPS behavior

The write-mask does not perform the normal write-masking function for this instruction. While it does enable/disable comparisons, it does not block updating of the destination; instead, if a write-mask bit is 0, the corresponding destination bit is set to 0. Nonetheless, the operation is similar enough so that it makes sense to use the usual write-mask notation. This mode of operation is desirable because the result will be used directly as a write-mask, rather than the normal case where the result is used with a separate write-mask that keeps the masked elements inactive.

Immediate Format

	Comparison Type	I_2	I_1	I_0
eq	Equal	0	0	0
lt	Less than	0	0	1
le	Less than or Equal	0	1	0
unord	Unordered	0	1	1
neq	Not Equal	1	0	0
nlt	Not Less than	1	0	1
nle	Not Less than or Equal	1	1	0
ord	Ordered	1	1	1

Operation

```
switch (IMM8[2:0]) {
    case 0: OP ← EQ; break;
    case 1: OP ← LT; break;
    case 2: OP ← LE; break;
    case 3: OP ← UNORD; break;
    case 4: OP ← NEQ; break;
    case 5: OP ← NLT; break;
    case 6: OP ← NLE; break;
    case 7: OP ← ORD; break;
}

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{f32}$(zmm2/$m_t$)
}

for (n = 0; n < 16; n++) {
  k2[n] = 0
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    k2[n] = (zmm1[i+31:i] OP tmpSrc2[i+31:i]) ? 1 : 0
  }
}
```

Instruction Pseudo-ops

Compilers and assemblers may implement the following pseudo-ops in addition to the standard instruction op:

Pseudo-Op	Implementation
vcmpeqps k2 {k1}, zmm1, S_f(zmm2/m_t)	vcmpps k2 {k1}, zmm1, S_f(zmm2/m_t), {eq}
vcmpltps k2 {k1}, zmm1, S_f(zmm2/m_t)	vcmpps k2 {k1}, zmm1, S_f(zmm2/m_t), {lt}
vcmpleps k2 {k1}, zmm1, S_f(zmm2/m_t)	vcmpps k2 {k1}, zmm1, S_f(zmm2/m_t), {le}
vcmpunordps k2 {k1}, zmm1, S_f(zmm2/m_t)	vcmpps k2 {k1}, zmm1, S_f(zmm2/m_t), {unord}
vcmpneqps k2 {k1}, zmm1, S_f(zmm2/m_t)	vcmpps k2 {k1}, zmm1, S_f(zmm2/m_t), {neq}
vcmpnltps k2 {k1}, zmm1, S_f(zmm2/m_t)	vcmpps k2 {k1}, zmm1, S_f(zmm2/m_t), {nlt}
vcmpnleps k2 {k1}, zmm1, S_f(zmm2/m_t)	vcmpps k2 {k1}, zmm1, S_f(zmm2/m_t), {nle}
vcmpordps k2 {k1}, zmm1, S_f(zmm2/m_t)	vcmpps k2 {k1}, zmm1, S_f(zmm2/m_t), {ord}

SIMD Floating-Point Exceptions

Invalid, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :

```
Not Applicable
```

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__mmask16   _mm512_cmpeq_ps_mask (__m512, __m512);
__mmask16   _mm512_mask_cmpeq_ps_mask (__mmask16, __m512, __m512);
__mmask16   _mm51_cmplt_ps_mask (__m512, __m512);
__mmask16   _mm512_mask_cmplt_ps_mask (__mmask16, __m512, __m512);
__mmask16   _mm512_cmple_ps_mask (__m512, __m512);
__mmask16   _mm512_mask_cmple_ps_mask (__mmask16, __m512, __m512);
__mmask16   _mm512_cmpunord_ps_mask (__m512, __m512);
__mmask16   _mm512_mask_cmpunord_ps_mask (__mmask16, __m512, __m512);
__mmask16   _mm512_cmpneq_ps_mask (__m512, __m512);
__mmask16   _mm512_mask_cmpneq_ps_mask (__mmask16, __m512, __m512);
__mmask16   _mm512_cmpnlt_ps_mask (__m512, __m512);
__mmask16   _mm512_mask_cmpnlt_ps_mask (__mmask16, __m512, __m512);
__mmask16   _mm512_cmpnle_ps_mask (__m512, __m512);
__mmask16   _mm512_mask_cmpnle_ps_mask (__mmask16, __m512, __m512);
__mmask16   _mm512_cmpord_ps_mask (__m512, __m512);
__mmask16   _mm512_mask_cmpord_ps_mask (__mmask16, __m512, __m512);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VCVTDQ2PD - Convert Int32 Vector to Float64 Vector

Opcode	Instruction	Description
MVEX.512.F3.0F.W0 E6 /r	vcvtdq2pd zmm1 {k1}, S_{i32}(zmm2/m_t)	Convert int32 vector S_{i32}(zmm2/m_t) to float64, and store the result in zmm1, under write-mask.

Description

Performs an element-by-element conversion from the int32 vector result of the swizzle/broadcast/conversion from memory or int32 vector zmm2 to a float64 vector . The result is written into float64 vector zmm1. The int32 source is read from either the lower half of the source operand (int32 vector zmm2), full memory source (8 elements, i.e. 256-bits) or the broadcast memory source.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc2[255:0] = zmm2[255:0]
} else {
   tmpSrc2[255:0] = SwizzUpConvLoad$_{i32}$(zmm2/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    j = 32*n
    zmm1[i+63:i] =
        CvtInt32ToFloat64(tmpSrc2[j+31:j])
  }
}
```

SIMD Floating-Point Exceptions

None.

Denormal Handling

Treat Input Denormals As Zeros :

`Not Applicable`

Flush Tiny Results To Zero :

`Not Applicable`

Memory Up-conversion: S$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	32
001	broadcast 1 element (x8)	[rax] {1to8}	4
010	broadcast 4 elements (x4)	[rax] {4to8}	16
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

_m512d	_mm512_cvtepi32lo_pd (__m512i);
_m512d	_mm512_mask_cvtepi32lo_pd (__m512d, __mmask8, __m512i);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to 4, 16 or 32-byte (depending on the swizzle broadcast).
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. This instruction does not support any SwizzUpConv involving data conversion. If SwizzUpConvMem function from memory is set to any value different than "no action", $\{1to8\}$ or$\{4to8\}$ then an Invalid Opcode fault is raised. Note that this rule only applies to memory conversions (register swizzles are allowed).

VCVTFXPNTDQ2PS - Convert Fixed Point Int32 Vector to Float32 Vector

Opcode	**Instruction**	**Description**
MVEX.512.0F3A.W0 CB /r ib	vcvtfxpntdq2ps zmm1 {k1}, S_{i32}(zmm2/m_t), $imm8$	Convert int32 vector S_{i32}(zmm2/m_t) to float32, and store the result in zmm1, using $imm8$, under write-mask.

Description

Performs an element-by-element conversion from the int32 vector result of the swizzle/broadcast/conversion from memory or int32 vector zmm2 to a float32 vector , then performs an optional adjustment to the exponent.

This instruction is write-masked, 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.

Immediate Format

Exponent Adjustment	value	I_7	I_6	I_5	I_4
0	2^0 (32.0 - no exponent adjustment)	0	0	0	0
4	2^4 (28.4)	0	0	0	1
5	2^5 (27.5)	0	0	1	0
8	2^8 (24.8)	0	0	1	1
16	2^{16} (16.16)	0	1	0	0
24	2^{24} (8.24)	0	1	0	1
31	2^{31} (1.31)	0	1	1	0
32	2^{32} (0.32)	0	1	1	1
reserved	*must UD*	1	x	x	x

Operation

```
expadj = IMM8[6:4]
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc2[511:0] = SwizzUpConvLoad$_{i32}$(zmm2/$m_t$)
}
```

```
for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] =
     CvtInt32ToFloat32(tmpSrc2[i+31:i], RoundingMode) / EXPADJ_TABLE[expadj]
  }
}
```

SIMD Floating-Point Exceptions

Precision.

Denormal Handling

Treat Input Denormals As Zeros :
`Not Applicable`

Flush Tiny Results To Zero :
`Not Applicable`

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

__m512	_mm512_cvtfxpnt_round_adjustepi32_ps(__m512i, int, _MM_EXP_ADJ_ENUM);
__m512	_mm512_mask_cvtfxpnt_round_adjustepi32_ps(__m512, __mmask16, __m512i, int, _MM_EXP_ADJ_ENUM);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.

If preceded by any REX, F0, F2, F3, or 66 prefixes.

VCVTFXPNTPD2DQ - Convert Float64 Vector to Fixed Point Int32 Vector

Opcode	Instruction	Description
MVEX.512.F2.0F3A.W1 E6 /r ib	vcvtfxpntpd2dq zmm1 {k1}, S_{f64}(zmm2/m_t), $imm8$	Convert float64 vector S_{f64}(zmm2/m_t) to int32, and store the result in zmm1, using $imm8$, under write-mask.

Description

Performs an element-by-element conversion and rounding from the float64 vector result of the swizzle/broadcast/conversion from memory or float64 vector zmm2 to a int32 vector . The int32 result is written into the lower half of the destination register zmm1; the other half of the destination is set to zero.

Out-of-range values are converted to the nearest representable value and that NaNs convert to 0, because this makes the calculation of Exp2 more efficient (avoiding problems with converting very large values to integers, where undetected incorrect values could otherwise result from overflow). Table 6.5 describes what should be the result when dealing with floating-point special number.

Input	Result
NaN	0
$+\infty$	INT_MAX
+0	0
-0	0
$-\infty$	INT_MIN

Table 6.5: Converting to integer special floating-point values behavior

This instruction is write-masked, 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.

Immediate Format

	Rounding Mode	I_1	I_0
rn	Round to Nearest (even)	0	0
rd	Round Down (Round toward Negative Infinity)	0	1
ru	Round Up (Round toward Positive Infinity)	1	0
rz	Round toward Zero	1	1

Operation

```
RoundingMode = IMM8[1:0]

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{f64}$(zmm2/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    j = 32*n
    zmm1[j+31:j] =
        CvtFloat64ToInt32(tmpSrc2[i+63:i], RoundingMode)
  }
}

zmm1[511:256] = 0
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Denormal Handling

Treat Input Denormals As Zeros :

```
        (MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
     Not Applicable
```

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

__m512i	_mm512_cvtfxpnt_roundpd_epi32lo(__m512d, int);
__m512i	_mm512_mask_cvtfxpnt_roundpd_epi32lo(__m512i, __mmask8, __m512d, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VCVTFXPNTPD2UDQ - Convert Float64 Vector to Fixed Point Uint32 Vector

Opcode	Instruction	Description
MVEX.512.F2.0F3A.W1 CA /r ib	vcvtfxpntpd2udq zmm1 {k1}, S_{f64}(zmm2/m_t), $imm8$	Convert float64 vector S_{f64}(zmm2/m_t) to uint32, and store the result in zmm1, using $imm8$, under write-mask.

Description

Performs an element-by-element conversion and rounding from the float64 vector result of the swizzle/broadcast/conversion from memory or float64 vector zmm2 to a uint32 vector . The uint32 result is written into the lower half of the destination register zmm1; the other half of the destination is set to zero.

Out-of-range values are converted to the nearest representable value and that NaNs convert to 0, because this makes the calculation of Exp2 more efficient (avoiding problems with converting very large values to integers, where undetected incorrect values could otherwise result from overflow). Table 6.6 describes what should be the result when dealing with floating-point special number.

Input	Result
NaN	0
$+\infty$	INT_MAX
+0	0
-0	0
$-\infty$	INT_MIN

Table 6.6: Converting to integer special floating-point values behavior

This instruction is write-masked, 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.

Immediate Format

	Rounding Mode	I_1	I_0
rn	Round to Nearest (even)	0	0
rd	Round Down (Round toward Negative Infinity)	0	1
ru	Round Up (Round toward Positive Infinity)	1	0
rz	Round toward Zero	1	1

Operation

```
RoundingMode = IMM8[1:0]

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{f64}$(zmm2/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    j = 32*n
    zmm1[j+31:j] =
        CvtFloat64ToUint32(tmpSrc2[i+63:i], RoundingMode)
  }
}

zmm1[511:256] = 0
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Denormal Handling

Treat Input Denormals As Zeros :

```
        (MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
     Not Applicable
```

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_cvtfxpnt_roundpd_epi32lo(__m512d, int);
__m512i    _mm512_mask_cvtfxpnt_roundpd_epi32lo(__m512i, __mmask8, __m512d, int);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VCVTFXPNTPS2DQ - Convert Float32 Vector to Fixed Point Int32 Vector

Opcode	**Instruction**	**Description**
MVEX.512.66.0F3A.W0 CB /r ib	vcvtfxpntps2dq zmm1 {k1}, S_{f32}(zmm2/m_t), $imm8$	Convert float32 vector S_{f32}(zmm2/m_t) to int32, and store the result in zmm1, using $imm8$, under write-mask.

Description

Performs an element-by-element conversion and rounding from the float32 vector result of the swizzle/broadcast/conversion from memory or float32 vector zmm2 to a int32 vector , with an optional exponent adjustment before the conversion.

Out-of-range values are converted to the nearest representable value and that NaNs convert to 0, because this makes the calculation of Exp2 more efficient (avoiding problems with converting very large values to integers, where undetected incorrect values could otherwise result from overflow). Table 6.7 describes what should be the result when dealing with floating-point special number.

Input	Result
NaN	0
$+\infty$	INT_MAX
+0	0
-0	0
$-\infty$	INT_MIN

Table 6.7: Converting to integer special floating-point values behavior

This instruction is write-masked, 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.

Immediate Format

	Rounding Mode	I_1	I_0
rn	Round to Nearest (even)	0	0
rd	Round Down (Round toward Negative Infinity)	0	1
ru	Round Up (Round toward Positive Infinity)	1	0
rz	Round toward Zero	1	1

Exponent Adjustment	value	I_7	I_6	I_5	I_4
0	2^0 (32.0 - no exponent adjustment)	0	0	0	0
4	2^4 (28.4)	0	0	0	1
5	2^5 (27.5)	0	0	1	0
8	2^8 (24.8)	0	0	1	1
16	2^{16} (16.16)	0	1	0	0
24	2^{24} (8.24)	0	1	0	1
31	2^{31} (1.31)	0	1	1	0
32	2^{32} (0.32)	0	1	1	1
reserved	*must UD*	1	x	x	x

Operation

```
RoundingMode = IMM8[1:0]
expadj = IMM8[6:4]

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{f32}$(zmm2/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] =
     CvtFloat32ToInt32(tmpSrc2[i+31:i] * EXPADJ_TABLE[expadj], RoundingMode)
  }
}
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)? YES : NO`

Flush Tiny Results To Zero :
`Not Applicable`

Memory Up-conversion: S_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S_{f32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i   _mm512_cvtfxpnt_round_adjustps_epi32(__m512, int, _MM_EXP_ADJ_ENUM);
__m512i   _mm512_mask_cvtfxpnt_round_adjustps_epi32( __m512i, __mmask16, __m512,
          int, _MM_EXP_ADJ_ENUM);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VCVTFXPNTPS2UDQ - Convert Float32 Vector to Fixed Point Uint32 Vector

Opcode	Instruction	Description
MVEX.512.66.0F3A.W0 CA /r ib	vcvtfxpntps2udq zmm1 {k1}, S_{f32}(zmm2/m_t), $imm8$	Convert float32 vector S_{f32}(zmm2/m_t) to uint32, and store the result in zmm1, using $imm8$, under write-mask.

Description

Performs an element-by-element conversion and rounding from the float32 vector result of the swizzle/broadcast/conversion from memory or float32 vector zmm2 to a uint32 vector , with an optional exponent adjustment before the conversion.

Out-of-range values are converted to the nearest representable value and that NaNs convert to 0, because this makes the calculation of Exp2 more efficient (avoiding problems with converting very large values to integers, where undetected incorrect values could otherwise result from overflow). Table 6.8 describes what should be the result when dealing with floating-point special number.

Input	Result
NaN	0
$+\infty$	INT_MAX
+0	0
-0	0
$-\infty$	INT_MIN

Table 6.8: Converting to integer special floating-point values behavior

This instruction is write-masked, 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.

Immediate Format

	Rounding Mode	I_1	I_0
rn	Round to Nearest (even)	0	0
rd	Round Down (Round toward Negative Infinity)	0	1
ru	Round Up (Round toward Positive Infinity)	1	0
rz	Round toward Zero	1	1

Exponent Adjustment	value	I_7	I_6	I_5	I_4
0	2^0 (32.0 - no exponent adjustment)	0	0	0	0
4	2^4 (28.4)	0	0	0	1
5	2^5 (27.5)	0	0	1	0
8	2^8 (24.8)	0	0	1	1
16	2^{16} (16.16)	0	1	0	0
24	2^{24} (8.24)	0	1	0	1
31	2^{31} (1.31)	0	1	1	0
32	2^{32} (0.32)	0	1	1	1
reserved	*must UD*	1	x	x	x

Operation

```
RoundingMode = IMM8[1:0]
expadj = IMM8[6:4]

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{f32}$(zmm2/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] =
     CvtFloat32ToUint32(tmpSrc2[i+31:i] * EXPADJ_TABLE[expadj], RoundingMode)
  }
}
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)? YES : NO`

Flush Tiny Results To Zero :
`Not Applicable`

Memory Up-conversion: S_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S_{f32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

__m512i	_mm512_cvtfxpnt_round_adjustps_epi32(__m512, int, _MM_EXP_ADJ_ENUM);
__m512i	_mm512_mask_cvtfxpnt_round_adjustps_epi32(__m512i, __mmask16, __m512, int, _MM_EXP_ADJ_ENUM);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VCVTFXPNTUDQ2PS - Convert Fixed Point Uint32 Vector to Float32 Vector

Opcode	Instruction	Description
MVEX.512.0F3A.W0 CA /r ib	vcvtfxpntudq2ps zmm1 {k1}, S_{i32}(zmm2/m_t), $imm8$	Convert uint32 vector S_{i32}(zmm2/m_t) to float32, and store the result in zmm1, using $imm8$, under write-mask.

Description

Performs an element-by-element conversion from the uint32 vector result of the swizzle/broadcast/conversion from memory or uint32 vector zmm2 to a float32 vector , then performs an optional adjustment to the exponent.

This instruction is write-masked, 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.

Immediate Format

Exponent Adjustment	value	I_7	I_6	I_5	I_4
0	2^0 (32.0 - no exponent adjustment)	0	0	0	0
4	2^4 (28.4)	0	0	0	1
5	2^5 (27.5)	0	0	1	0
8	2^8 (24.8)	0	0	1	1
16	2^{16} (16.16)	0	1	0	0
24	2^{24} (8.24)	0	1	0	1
31	2^{31} (1.31)	0	1	1	0
32	2^{32} (0.32)	0	1	1	1
reserved	*must UD*	1	x	x	x

Operation

```
expadj = IMM8[6:4]
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   RoundingMode = MXCSR.RC
```

```
    tmpSrc2[511:0] = SwizzUpConvLoad$_{i32}$(zmm2/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] =
     CvtUint32ToFloat32(tmpSrc2[i+31:i], RoundingMode) / EXPADJ_TABLE[expadj]
  }
}
```

SIMD Floating-Point Exceptions

Precision.

Denormal Handling

Treat Input Denormals As Zeros :

`Not Applicable`

Flush Tiny Results To Zero :

`Not Applicable`

Memory Up-conversion: S$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VCVTPD2PS - Convert Float64 Vector to Float32 Vector

Opcode	Instruction	Description
MVEX.512.66.0F.W1 5A /r	vcvtpd2ps zmm1 {k1}, S_{f64}(zmm2/m_t)	Convert float64 vector S_{f64}(zmm2/m_t) to float32, and store the result in zmm1, under write-mask.

Description

Performs an element-by-element conversion and rounding from the float64 vector result of the swizzle/broadcast/conversion from memory or float64 vector zmm2 to a float32 vector . The result is written into float32 vector zmm1. The float32 result is written into the lower half of the destination register zmm1; the other half of the destination is set to zero.

Input	Result
NaN	Quietized NaN. Copy leading bits of float64 significand
$+\infty$	$+\infty$
+0	$+0$
-0	-0
$-\infty$	$-\infty$

Table 6.9: Converting float64 to float32 special values behavior

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc2[511:0] = SwizzUpConvLoad$_{f64}$(zmm2/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    j = 32*n
    zmm1[j+31:j] =
```

```
        CvtFloat64ToFloat32(tmpSrc2[i+63:i], RoundingMode)
  }
}

zmm1[511:256] = 0
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
        (MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
     (MXCSR.FZ)? YES : NO
```

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S_{f64}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_cvtpd_pslo (__m512d);
__m512    _mm512_mask_cvtpd_pslo (__m512d, __mmask8, __m512d);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.

	If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VCVTPS2PD - Convert Float32 Vector to Float64 Vector

Opcode	Instruction	Description
MVEX.512.0F.W0 5A /r	vcvtps2pd zmm1 {k1}, S_{f32}(zmm2/m_t)	Convert float32 vector S_{f32}(zmm2/m_t) to float64, and store the result in zmm1, under write-mask.

Description

Performs an element-by-element conversion and rounding from the float32 vector result of the swizzle/broadcast/conversion from memory or float32 vector zmm2 to a float64 vector . The result is written into float64 vector zmm1. The float32 source is read from either the lower half of the source operand (float32 vector zmm2), full memory source (8 elements, i.e. 256-bits) or the broadcast memory source.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc2[255:0] = zmm2[255:0]
} else {
   tmpSrc2[255:0] = SwizzUpConvLoad$_{f32}$(zmm2/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    j = 32*n
    zmm1[i+63:i] =
        CvtFloat32ToFloat64(tmpSrc2[j+31:j])
  }
}
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :

```
Not Applicable
```

Memory Up-conversion: S_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	32
001	broadcast 1 element (x8)	[rax] {1to8}	4
010	broadcast 4 elements (x4)	[rax] {4to8}	16
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S_{f32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

_m512d	_mm512_cvtpslo_pd (__m512);
_m512d	_mm512_mask_cvtpslo_pd (__m512d, __mmask8, __m512);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to 4, 16 or 32-byte (depending on the swizzle broadcast).
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. This instruction does not support any SwizzUpConv involving data conversion. If SwizzUpConvMem function from memory is set to any value different than "no action", $\{1to8\}$ or $\{4to8\}$ then an Invalid Opcode fault is raised. Note that this rule only applies to memory conversions (register swizzles are allowed).

VCVTUDQ2PD - Convert Uint32 Vector to Float64 Vector

Opcode	Instruction	Description
MVEX.512.F3.0F.W0 7A /r	vcvtudq2pd zmm1 {k1}, S_{i32}(zmm2/m_t)	Convert uint32 vector S_{i32}(zmm2/m_t) to float64, and store the result in zmm1, under write-mask.

Description

Performs an element-by-element conversion from the uint32 vector result of the swizzle/broadcast/conversion from memory or uint32 vector zmm2 to a float64 vector . The result is written into float64 vector zmm1. The uint32 source is read from either the lower half of the source operand (uint32 vector zmm2), full memory source (8 elements, i.e. 256-bits) or the broadcast memory source.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc2[255:0] = zmm2[255:0]
} else {
   tmpSrc2[255:0] = SwizzUpConvLoad$_{i32}$(zmm2/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    j = 32*n
    zmm1[i+63:i] =
        CvtUint32ToFloat64(tmpSrc2[j+31:j])
  }
}
```

SIMD Floating-Point Exceptions

None.

Denormal Handling

Treat Input Denormals As Zeros :

`Not Applicable`

Flush Tiny Results To Zero :

`Not Applicable`

Memory Up-conversion: S$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	32
001	broadcast 1 element (x8)	[rax] {1to8}	4
010	broadcast 4 elements (x4)	[rax] {4to8}	16
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
_m512d    _mm512_cvtepu32lo_pd (__m512i);
_m512d    _mm512_mask_cvtepu32lo_pd (__m512d, __mmask8, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to 4, 16 or 32-byte (depending on the swizzle broadcast).
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. This instruction does not support any SwizzUpConv involving data conversion. If SwizzUpConvMem function from memory is set to any value different than "no action", $\{1to8\}$ or $\{4to8\}$ then an Invalid Opcode fault is raised. Note that this rule only applies to memory conversions (register swizzles are allowed).

VEXP223PS - Base-2 Exponential Calculation of Float32 Vector

Opcode	**Instruction**	**Description**
MVEX.512.66.0F38.W0 C8 /r	vexp223ps zmm1 {k1}, zmm2/m_t	Calculate the approx. exp2 from int32 vector zmm2/m_t and store the result in zmm1, under write-mask.

Description

Computes the element-by-element base-2 exponential computation of the int32 vector on memory or int32 vector zmm2 with 0.99ULP (relative error). Input int32 values are considered as fixed point numbers with a fraction offset of 24 bits (i.e. 8 MSBs correspond to sign and integer part; 24 LSBs correspond to fractional part). The result is written into float32 vector zmm1.

exp2 of a FP input value is computed as a two-instruction sequence:

1. vcvtfxpntps2dq (with exponent adjustment, so that destination format is 32b, with 8b for integer part and 24b for fractional part)
2. vexp223ps

All overflows are captured by the combination of the saturating behavior of vcvtfxpntps2dq instruction and the detection of MAX_INT/MIN_INT by the vexp223ps instruction. Tiny input numbers are quietly flushed to the fixed-point value 0 by the vcvtfxpntps2dq instruction, which produces an overall output $exp2(0) = 1.0f$.

The overall behavior of the two-instruction sequence is the following:

- $-\infty$ returns $+0.0f$
- $\pm 0.0f$ returns $1.0f$ (exact result)
- $+\infty$ returns +∞ (#Overflow)
- NaN returns $1.0f$ (#Invalid)
- n, where n is an integral value returns 2^n (exact result)

Input	Result	Comments
MIN_INT	+0.0f	
MAX_INT	$+\infty$	Raise #O flag

Table 6.10: vexp2_1ulp() special int values behavior

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
tmpSrc2[511:0] = zmm2/$m_t$

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags()    // SAE
}

for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] = exp2_1ulp(tmpSrc2[i+31:i])
  }
}
```

SIMD Floating-Point Exceptions

Overflow.

Denormal Handling

Treat Input Denormals As Zeros :
`Not Applicable`

Flush Tiny Results To Zero :
`YES`

Register Swizzle

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	reserved	N/A
010	reserved	N/A
011	reserved	N/A
100	reserved	N/A
101	reserved	N/A
110	reserved	N/A
111	reserved	N/A

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_exp223_ps (__m512i);
__m512    _mm512_mask_exp223_ps (__m512, __mmask16, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.
	This instruction does not support any SwizzUpConv different from the default value (no broadcast, no conversion). If SwizzUpConv function is set to any value different than "no action", then an Invalid Opcode fault is raised. This includes register swizzles.

VFIXUPNANPD - Fix Up Special Float64 Vector Numbers With NaN Passthrough

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W1 55 /r	vfixupnanpd zmm1 {k1}, zmm2, S_{i64}(zmm3/m_t)	Fix up, with NaN passthrough, special numbers in float64 vector zmm1, float64 vector zmm2 and int64 vector S_{i64}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element fix-up of various real and special number types in the float64 vector zmm2 using the 21-bit table values from the result of the swizzle/broadcast/conversion process on memory or int64 vector zmm3. The result is merged into float64 vector zmm1. Unlike in *vfixuppd*, source NaN values are passed-through as quietized values. Note that, also unlike in *vfixup*, this quietization translates into a #IE exception flag being reported for input SNaNs.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source, 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. However, evaluating rcp via $2x - ax^2$ (Newton-Raphson), where $x = approx(1/0) = \infty$, incorrectly yields NaN. To deal with this, vfixupps can be used after the N-R reciprocal sequence to set the result to ∞ when the input is 0.

Denormal inputs must be treated as zeros of the same sign if DAZ is enabled.

Note that NO_CHANGE_TOKEN leaves the destination (output) unchanged. This means that if the destination is a denormal, its value is not flushed to 0.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
enum TOKEN_TYPE
{
    NO_CHANGE_TOKEN  = 0,
```

```
    NEG_INF_TOKEN    = 1,
    NEG_ZERO_TOKEN   = 2,
    POS_ZERO_TOKEN   = 3,
    POS_INF_TOKEN    = 4,
    NAN_TOKEN        = 5,
    MAX_DOUBLE_TOKEN = 6,
    MIN_DOUBLE_TOKEN = 7,
}

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpzmm3[511:0] = zmm3[511:0]
} else {
   tmpzmm3[511:0] = SwizzUpConvLoad$_{i64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if (k1[n] != 0) {
    i = 64*n
    tsrc[63:0] = zmm2[i+63:i]

    if (IsNaN(tsrc[63:0])
    {
       zmm1[i+63:i] = QNaN(zmm2[i+63:i])
    }
    else
    {
       // tmp is an int value
       if      (tsrc[63:0] == -inf)     tmp = 0
       else if (tsrc[63:0] < 0)     tmp = 1
       else if (tsrc[63:0] == -0)  tmp = 2
       else if (tsrc[63:0] == +0)  tmp = 3
       else if (tsrc[63:0] == inf) tmp = 5
       else /*  tsrc[63:0] > 0  */ tmp = 4

       table[20:0] = tmpzmm3[i+63:i]
       token = table[(tmp*3)+2: tmp*3]  // table is viewed as one 21-bit
                                        // little-endian value.
                                        // token is an int value
                                        // the 7th entry is unused

       // float64 result
       if (token == NEG_INF_TOKEN)         zmm1[i+63:i] = -inf
       else if (token == NEG_ZERO_TOKEN)   zmm1[i+63:i] = -0
       else if (token == POS_ZERO_TOKEN)   zmm1[i+63:i] = +0
       else if (token == POS_INF_TOKEN)    zmm1[i+63:i] = +inf
      else if (token == NAN_TOKEN)        zmm1[i+63:i] = QNaN_indefinite
       else if (token == MAX_DOUBLE_TOKEN) zmm1[i+63:i] = NMAX
       else if (token == MIN_DOUBLE_TOKEN) zmm1[i+63:i] = -NMAX
     else if (token == NO_CHANGE_TOKEN)  { /* zmm1[i+63:i] remains unchanged */ }
    }
  }
```

```
}
```

SIMD Floating-Point Exceptions

Invalid.

Denormal Handling

Treat Input Denormals As Zeros :

```
        (MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
    NO
```

Memory Up-conversion: S$_{i64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{i64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_fixupnan_pd (__m512d, __m512d, __m512i);
__m512d    _mm512_mask_fixupnan_pd (__m512d, __mmask8, __m512d, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFIXUPNANPS - Fix Up Special Float32 Vector Numbers With NaN Passthrough

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 55 /r	vfixupnanps zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Fix up, with NaN passthrough, special numbers in float32 vector zmm1, float32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element fix-up of various real and special number types in the float32 vector zmm2 using the 21-bit table values from the result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3. The result is merged into float32 vector zmm1. Unlike in *vfixupps*, source NaN values are passed-through as quietized values. Note that, also unlike in *vfixup*, this quietization translates into a #IE exception flag being reported for input SNaNs.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source, 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. However, evaluating rcp via $2x - ax^2$ (Newton-Raphson), where $x = approx(1/0) = \infty$, incorrectly yields NaN. To deal with this, vfixupps can be used after the N-R reciprocal sequence to set the result to ∞ when the input is 0.

Denormal inputs must be treated as zeros of the same sign if DAZ is enabled.

Note that NO_CHANGE_TOKEN leaves the destination (output) unchanged. This means that if the destination is a denormal, its value is not flushed to 0.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
enum TOKEN_TYPE
{
    NO_CHANGE_TOKEN  = 0,
```

```
    NEG_INF_TOKEN    = 1,
    NEG_ZERO_TOKEN   = 2,
    POS_ZERO_TOKEN   = 3,
    POS_INF_TOKEN    = 4,
    NAN_TOKEN        = 5,
    MAX_FLOAT_TOKEN  = 6,
    MIN_FLOAT_TOKEN  = 7,
}

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpzmm3[511:0] = zmm3[511:0]
} else {
   tmpzmm3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    tsrc[31:0] = zmm2[i+31:i]

    if (IsNaN(tsrc[31:0])
    {
       zmm1[i+31:i] = QNaN(zmm2[i+31:i])
    }
    else
    {
       // tmp is an int value
       if      (tsrc[31:0] == -inf)     tmp = 0
       else if (tsrc[31:0] < 0)     tmp = 1
       else if (tsrc[31:0] == -0)  tmp = 2
       else if (tsrc[31:0] == +0)  tmp = 3
       else if (tsrc[31:0] == inf) tmp = 5
       else /*  tsrc[31:0] > 0  */ tmp = 4

       table[20:0] = tmpzmm3[i+31:i]
       token = table[(tmp*3)+2: tmp*3]  // table is viewed as one 21-bit
                                        // little-endian value.
                                        // token is an int value
                                        // the 7th entry is unused

       // float32 result
       if (token == NEG_INF_TOKEN)         zmm1[i+31:i] = -inf
       else if (token == NEG_ZERO_TOKEN)   zmm1[i+31:i] = -0
       else if (token == POS_ZERO_TOKEN)   zmm1[i+31:i] = +0
       else if (token == POS_INF_TOKEN)    zmm1[i+31:i] = +inf
      else if (token == NAN_TOKEN)        zmm1[i+31:i] = QNaN_indefinite
       else if (token == MAX_FLOAT_TOKEN)  zmm1[i+31:i] = NMAX
       else if (token == MIN_FLOAT_TOKEN)  zmm1[i+31:i] = -NMAX
     else if (token == NO_CHANGE_TOKEN)  { /* zmm1[i+31:i] remains unchanged */ }
    }
  }
```

```
}
```

SIMD Floating-Point Exceptions

Invalid.

Denormal Handling

Treat Input Denormals As Zeros :
```
(MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :
```
NO
```

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_fixupnan_ps (__m512, __m512, __m512i);
__m512    _mm512_mask_fixupnan_ps (__m512, __mmask16, __m512, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFMADD132PD - Multiply Destination By Second Source and Add To First Source Float64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W1 98 /r	vfmadd132pd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Multiply float64 vector zmm1 and float64 vector S_{f64}(zmm3/m_t), add the result to float64 vector zmm2, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between float64 vector zmm1 and the float64 vector result of the swizzle/broadcast/conversion process on memory or vector float64 zmm3, then adds the result to float64 vector zmm2. The final sum is written into float64 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = zmm1[i+63:i] * tmpSrc3[i+63:i] + zmm2[i+63:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
```
(MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :
```
(MXCSR.FZ)?  YES : NO
```

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_fmadd_pd (__m512d, __m512d, __m512d);
__m512d    _mm512_mask_fmadd_pd (__m512d, __mmask8, __m512d, __m512d);
__m512d    _mm512_mask3_fmadd_pd (__m512d, __m512d, __m512d, __mmask8);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is

	in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFMADD132PS - Multiply Destination By Second Source and Add To First Source Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 98 /r	vfmadd132ps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Multiply float32 vector zmm1 and float32 vector S_{f32}(zmm3/m_t), add the result to float32 vector zmm2, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between float32 vector zmm1 and the float32 vector result of the swizzle/broadcast/conversion process on memory or vector float32 zmm3, then adds the result to float32 vector zmm2. The final sum is written into float32 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = zmm1[i+31:i] * tmpSrc3[i+31:i] + zmm2[i+31:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :

```
(MXCSR.FZ)? YES : NO
```

Memory Up-conversion: S_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S_{f32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512   _mm512_fmadd_ps (__m512, __m512, __m512);
__m512   _mm512_mask_fmadd_ps (__m512, __mmask16, __m512, __m512);
__m512   _mm512_mask3_fmadd_ps (__m512, __m512, __m512, __mmask16);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFMADD213PD - Multiply First Source By Destination and Add Second Source Float64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W1 A8 /r	vfmadd213pd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Multiply float64 vector zmm2 and float64 vector zmm1, add the result to float64 vector S_{f64}(zmm3/m_t), and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between float64 vector zmm2 and float64 vector zmm1 and then adds the result to the float64 vector result of the swizzle/broadcast/conversion process on memory or vector float64 zmm3. The final sum is written into float64 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = zmm2[i+63:i] * zmm1[i+63:i] + tmpSrc3[i+63:i]
```

```
    }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
(MXCSR.FZ)?  YES : NO
```

Memory Up-conversion: S_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_fmadd_pd (__m512d, __m512d, __m512d);
__m512d    _mm512_mask_fmadd_pd (__m512d, __mmask8, __m512d, __m512d);
__m512d    _mm512_mask3_fmadd_pd (__m512d, __m512d, __m512d, __mmask8);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is

	in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFMADD213PS - Multiply First Source By Destination and Add Second Source Float32 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W0 A8 /r	vfmadd213ps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Multiply float32 vector zmm2 and float32 vector zmm1, add the result to float32 vector S_{f32}(zmm3/m_t), and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between float32 vector zmm2 and float32 vector zmm1 and then adds the result to the float32 vector result of the swizzle/broadcast/conversion process on memory or vector float32 zmm3. The final sum is written into float32 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = zmm2[i+31:i] * zmm1[i+31:i] + tmpSrc3[i+31:i]
```

```
        }
    }
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
    (MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :

```
    (MXCSR.FZ)? YES : NO
```

Memory Up-conversion: S_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_fmadd_ps (__m512, __m512, __m512);
__m512    _mm512_mask_fmadd_ps (__m512, __mmask16, __m512, __m512);
__m512    _mm512_mask3_fmadd_ps (__m512, __m512, __m512, __mmask16);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.

#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFMADD231PD - Multiply First Source By Second Source and Add To Destination Float64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W1 B8 /r	vfmadd231pd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Multiply float64 vector zmm2 and float64 vector S_{f64}(zmm3/m_t), add the result to float64 vector zmm1, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between float64 vector zmm2 and the float64 vector result of the swizzle/broadcast/conversion process on memory or vector float64 zmm3, then adds the result to float64 vector zmm1. The final sum is written into float64 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = zmm2[i+63:i] * tmpSrc3[i+63:i] + zmm1[i+63:i]
  }
```

```
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
(MXCSR.FZ)?  YES : NO
```

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_fmadd_pd (__m512d, __m512d, __m512d);
__m512d    _mm512_mask_fmadd_pd (__m512d, __mmask8, __m512d, __m512d);
__m512d    _mm512_mask3_fmadd_pd (__m512d, __m512d, __m512d, __mmask8);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is

	in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFMADD231PS - Multiply First Source By Second Source and Add To Destination Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 B8 /r	vfmadd231ps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Multiply float32 vector zmm2 and float32 vector S_{f32}(zmm3/m_t), add the result to float32 vector zmm1, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between float32 vector zmm2 and the float32 vector result of the swizzle/broadcast/conversion process on memory or vector float32 zmm3, then adds the result to float32 vector zmm1. The final sum is written into float32 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = zmm2[i+31:i] * tmpSrc3[i+31:i] + zmm1[i+31:i]
  }
```

```
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
```
      (MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :
```
      (MXCSR.FZ)? YES : NO
```

Memory Up-conversion: S_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_fmadd_ps (__m512, __m512, __m512);
__m512    _mm512_mask_fmadd_ps (__m512, __mmask16, __m512, __m512);
__m512    _mm512_mask3_fmadd_ps (__m512, __m512, __m512, __mmask16);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.

#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFMADD233PS - Multiply First Source By Specially Swizzled Second Source and Add To Second Source Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 A4 /r	vfmadd233ps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Multiply float32 vector zmm2 by certain elements of float32 vector S_{f32}(zmm3/m_t), add the result to certain elements of S_{f32}(zmm3/m_t), and store the final result in zmm1, under write-mask.

Description

This instruction is built around the concept of 4-element sets, of which there are four: elements 0-3, 4-7, 8-11, and 12-15. If we refer to the float32 vector result of the broadcast (no conversion is supported) process on memory or the float32 vector zmm3 (no swizzle is supported) as t3, then:

Each element 0-3 of float32 vector zmm2 is multiplied by element 1 of t3, the result is added to element 0 of t3, and the final sum is written into the corresponding element 0-3 of float32 vector zmm1.

Each element 4-7 of float32 vector zmm2 is multiplied by element 5 of t3, the result is added to element 4 of t3, and the final sum is written into the corresponding element 4-7 of float32 vector zmm1.

Each element 8-11 of float32 vector zmm2 is multiplied by element 9 of t3, the result is added to element 8 of t3, and the final sum is written into the corresponding element 8-11 of float32 vector zmm1.

Each element 12-15 of float32 vector zmm2 is multiplied by element 13 of t3, the result is added to element 12 of t3, and the final sum is written into the corresponding element 12-15 of float32 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded.

This instruction makes it possible to perform scale and bias in a single instruction without needing to have either scale or bias already loaded in a register. This saves one vector load for each interpolant, representing around ten percent of shader instructions.

For structure-of-arrays (SOA) operation, this instruction is intended to be used with the $\{4to16\}$ broadcast on src2, allowing all 16 scale and biases to be identical. For array-of-structures (AOS) vec4 operations, no broadcast is used, allowing four different scales and biases, one for each vec4.

No conversion or swizzling is supported for this instruction. However, all broadcasts except $\{1to16\}$ are supported (i.e. 16to16 and 4to16).

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    base = ( n & ~0x03 ) * 32
    scale[31:0] = tmpSrc3[base+63:base+32]
    bias[31:0] = tmpSrc3[base+31:base]
    // float32 operation
    zmm1[i+31:i] = zmm2[i+31:i] * scale[31:0] + bias[31:0]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :

```
(MXCSR.FZ)? YES : NO
```

Memory Up-conversion: S_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	reserved	N/A	N/A
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S_{f32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	reserved	N/A
010	reserved	N/A
011	reserved	N/A
100	reserved	N/A
101	reserved	N/A
110	reserved	N/A
111	reserved	N/A

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_fmadd233_ps (__m512, __m512);
__m512    _mm512_mask_fmadd233_ps (__m512, __mmask16, __m512, __m512);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to 16 or 64-byte (depending on the swizzle broadcast).
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. This instruction does not support any SwizzUpConv involving data conversion, register swizzling or $\{1to16\}$ broadcast. If SwizzUpConv function is set to any value different than "no action" or $\{4to16\}$ then an Invalid Opcode fault is raised

VFMSUB132PD - Multiply Destination By Second Source and Subtract First Source Float64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W1 9A /r	vfmsub132pd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Multiply float64 vector zmm1 and float64 vector S_{f64}(zmm3/m_t), subtract float64 vector zmm2 from the result, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication of float64 vector zmm1 and the float64 vector result of the swizzle/broadcast/conversion process on memory or vector float64 zmm3, then subtracts float64 vector zmm2 from the result. The final result is written into float64 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = zmm1[i+63:i] * tmpSrc3[i+63:i] - zmm2[i+63:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
(MXCSR.FZ)?  YES : NO
```

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_fmsub_pd (__m512d, __m512d, __m512d);
__m512d    _mm512_mask_fmsub_pd (__m512d, __mmask8, __m512d, __m512d);
__m512d    _mm512_mask3_fmsub_pd (__m512d, __m512d, __m512d, __mmask8);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is

	in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFMSUB132PS - Multiply Destination By Second Source and Subtract First Source Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 9A /r	vfmsub132ps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Multiply float32 vector zmm1 and float32 vector S_{f32}(zmm3/m_t), subtract float32 vector zmm2 from the result, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication of float32 vector zmm1 and the float32 vector result of the swizzle/broadcast/conversion process on memory or vector float32 zmm3, then subtracts float32 vector zmm2 from the result. The final result is written into float32 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = zmm1[i+31:i] * tmpSrc3[i+31:i] - zmm2[i+31:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :

```
(MXCSR.FZ)? YES : NO
```

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_fmsub_ps (__m512, __m512, __m512);
__m512    _mm512_mask_fmsub_ps (__m512, __mmask16, __m512, __m512);
__m512    _mm512_mask3_fmsub_ps (__m512, __m512, __m512, __mmask16);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFMSUB213PD - Multiply First Source By Destination and Subtract Second Source Float64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W1 AA /r	vfmsub213pd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Multiply float64 vector zmm2 and float64 vector zmm1, subtract float64 vector S_{f64}(zmm3/m_t) from the result, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication of float64 vector zmm2 and float64 vector zmm1, then subtracts the float64 vector result of the swizzle/broadcast/conversion process on memory or vector float64 zmm3 from the result. The final result is written into float64 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = zmm2[i+63:i] * zmm1[i+63:i] - tmpSrc3[i+63:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
```
(MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :
```
(MXCSR.FZ)?  YES : NO
```

Memory Up-conversion: S_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_fmsub_pd (__m512d, __m512d, __m512d);
__m512d    _mm512_mask_fmsub_pd (__m512d, __mmask8, __m512d, __m512d);
__m512d    _mm512_mask3_fmsub_pd (__m512d, __m512d, __m512d, __mmask8);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is

	in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFMSUB213PS - Multiply First Source By Destination and Subtract Second Source Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 AA /r	vfmsub213ps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Multiply float32 vector zmm2 and float32 vector zmm1, subtract float32 vector S_{f32}(zmm3/m_t) from the result, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication of float32 vector zmm2 and float32 vector zmm1, then subtracts the float32 vector result of the swizzle/broadcast/conversion process on memory or vector float32 zmm3 from the result. The final result is written into float32 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = zmm2[i+31:i] * zmm1[i+31:i] - tmpSrc3[i+31:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :

```
(MXCSR.FZ)? YES : NO
```

Memory Up-conversion: S_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S_{f32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_fmsub_ps (__m512, __m512, __m512);
__m512    _mm512_mask_fmsub_ps (__m512, __mmask16, __m512, __m512);
__m512    _mm512_mask3_fmsub_ps (__m512, __m512, __m512, __mmask16);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFMSUB231PD - Multiply First Source By Second Source and Subtract Destination Float64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W1 BA /r	vfmsub231pd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Multiply float64 vector zmm2 and float64 vector S_{f64}(zmm3/m_t), subtract float64 vector zmm1 from the result, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication of float32 vector zmm2 and the float32 vector result of the swizzle/broadcast/conversion process on memory or vector float32 zmm3, then subtracts float32 vector zmm1 from the result. The final result is written into float32 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = zmm2[i+63:i] * tmpSrc3[i+63:i] - zmm1[i+63:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
        (MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
        (MXCSR.FZ)?  YES : NO
```

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_fmsub_pd (__m512d, __m512d, __m512d);
__m512d    _mm512_mask_fmsub_pd (__m512d, __mmask8, __m512d, __m512d);
__m512d    _mm512_mask3_fmsub_pd (__m512d, __m512d, __m512d, __mmask8);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is

	in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFMSUB231PS - Multiply First Source By Second Source and Subtract Destination Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 BA /r	vfmsub231ps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Multiply float32 vector zmm2 and float32 vector S_{f32}(zmm3/m_t), subtract float32 vector zmm1 from the result, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication of float32 vector zmm2 and the float32 vector result of the swizzle/broadcast/conversion process on memory or vector float32 zmm3, then subtracts float32 vector zmm1 from the result. The final result is written into float32 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = zmm2[i+31:i] * tmpSrc3[i+31:i] - zmm1[i+31:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :

```
(MXCSR.FZ)? YES : NO
```

Memory Up-conversion: S_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S_{f32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_fmsub_ps (__m512, __m512, __m512);
__m512    _mm512_mask_fmsub_ps (__m512, __mmask16, __m512, __m512);
__m512    _mm512_mask3_fmsub_ps (__m512, __m512, __m512, __mmask16);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFNMADD132PD - Multiply Destination By Second Source and Subtract From First Source Float64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W1 9C /r	vfnmadd132pd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Multiply float64 vector zmm1 and float64 vector S_{f64}(zmm3/m_t), negate, and add the result to float64 vector zmm2, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication of float64 vector zmm2 and the float64 vector result of the swizzle/broadcast/conversion process on memory or vector float64 zmm3, then subtracts the result from float64 vector zmm1. The final result is written into float64 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
```

```
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = -(zmm1[i+63:i] * tmpSrc3[i+63:i]) + zmm2[i+63:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)? YES : NO`

Flush Tiny Results To Zero :
`(MXCSR.FZ)? YES : NO`

Memory Up-conversion: S_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S_{f64}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_fnmadd_pd (__m512d, __m512d, __m512d);
__m512d    _mm512_mask_fnmadd_pd (__m512d, __mmask8, __m512d, __m512d);
__m512d    _mm512_mask3_fnmadd_pd (__m512d, __m512d, __m512d, __mmask8);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is

	in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFNMADD132PS - Multiply Destination By Second Source and Subtract From First Source Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 9C /r	vfnmadd132ps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Multiply float32 vector zmm1 and float32 vector S_{f32}(zmm3/m_t), negate, and add the result to float32 vector zmm2, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication of float32 vector zmm2 and the float32 vector result of the swizzle/broadcast/conversion process on memory or vector float32 zmm3, then subtracts the result from float32 vector zmm1. The final result is written into float32 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
```

```
        i = 32*n
        // float32 operation
        zmm1[i+31:i] = -(zmm1[i+31:i] * tmpSrc3[i+31:i]) + zmm2[i+31:i]
      }
    }
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)? YES : NO`

Flush Tiny Results To Zero :
`(MXCSR.FZ)? YES : NO`

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512   _mm512_fnmadd_ps (__m512, __m512, __m512);
__m512   _mm512_mask_fnmadd_ps (__m512, __mmask16, __m512, __m512);
__m512   _mm512_mask3_fnmadd_ps (__m512, __m512, __m512, __mmask16);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.

#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFNMADD213PD - Multiply First Source By Destination and Subtract From Second Source Float64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W1 AC /r	vfnmadd213pd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Multiply float64 vector zmm2 and float64 vector zmm1, negate, and add the result to float64 vector S_{f64}(zmm3/m_t), and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication of float64 vector zmm1 and the float64 vector result of the swizzle/broadcast/conversion process on memory or vector float64 zmm3, then subtracts the result from float64 vector zmm2. The final result is written into float64 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
```

```
      // float64 operation
      zmm1[i+63:i] = -(zmm2[i+63:i] * zmm1[i+63:i]) + tmpSrc3[i+63:i]
    }
  }
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
        (MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
        (MXCSR.FZ)?  YES : NO
```

Memory Up-conversion: S_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d   _mm512_fnmadd_pd (__m512d, __m512d, __m512d);
__m512d   _mm512_mask_fnmadd_pd (__m512d, __mmask8, __m512d, __m512d);
__m512d   _mm512_mask3_fnmadd_pd (__m512d, __m512d, __m512d, __mmask8);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is

	in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFNMADD213PS - Multiply First Source By Destination and Subtract From Second Source Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 AC /r	vfnmadd213ps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Multiply float32 vector zmm2 and float32 vector zmm1, negate, and add the result to float32 vector S_{f32}(zmm3/m_t), and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication of float32 vector zmm1 and the float32 vector result of the swizzle/broadcast/conversion process on memory or vector float32 zmm3, then subtracts the result from float32 vector zmm2. The final result is written into float32 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
```

```
        // float32 operation
        zmm1[i+31:i] = -(zmm2[i+31:i] * zmm1[i+31:i]) + tmpSrc3[i+31:i]
    }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
```
    (MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :
```
    (MXCSR.FZ)? YES : NO
```

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_fnmadd_ps (__m512, __m512, __m512);
__m512    _mm512_mask_fnmadd_ps (__m512, __mmask16, __m512, __m512);
__m512    _mm512_mask3_fnmadd_ps (__m512, __m512, __m512, __mmask16);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFNMADD231PD - Multiply First Source By Second Source and Subtract From Destination Float64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W1 BC /r	vfnmadd231pd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Multiply float64 vector zmm2 and float64 vector S_{f64}(zmm3/m_t), negate, and add the result to float64 vector zmm1, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication of float64 vector zmm2 and float64 vector zmm1, then subtracts the result from the float64 vector result of the swizzle/broadcast/conversion process on memory or vector float64 zmm3. The final result is written into float64 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
```

```
        i = 64*n
        // float64 operation
        zmm1[i+63:i] = -(zmm2[i+63:i] * tmpSrc3[i+63:i]) + zmm1[i+63:i]
      }
    }
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
        (MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
        (MXCSR.FZ)?  YES : NO
```

Memory Up-conversion: S_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_fnmadd_pd (__m512d, __m512d, __m512d);
__m512d    _mm512_mask_fnmadd_pd (__m512d, __mmask8, __m512d, __m512d);
__m512d    _mm512_mask3_fnmadd_pd (__m512d, __m512d, __m512d, __mmask8);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is

	in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFNMADD231PS - Multiply First Source By Second Source and Subtract From Destination Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 BC /r	vfnmadd231ps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Multiply float32 vector zmm2 and float32 vector S_{f32}(zmm3/m_t), negate, and add the result to float32 vector zmm1, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication of float32 vector zmm2 and float32 vector zmm1, then subtracts the result from the float32 vector result of the swizzle/broadcast/conversion process on memory or vector float32 zmm3. The final result is written into float32 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
```

```
      i = 32*n
      // float32 operation
      zmm1[i+31:i] = -(zmm2[i+31:i] * tmpSrc3[i+31:i]) + zmm1[i+31:i]
    }
  }
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)? YES : NO`

Flush Tiny Results To Zero :
`(MXCSR.FZ)? YES : NO`

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512   _mm512_fnmadd_ps (__m512, __m512, __m512);
__m512   _mm512_mask_fnmadd_ps (__m512, __mmask16, __m512, __m512);
__m512   _mm512_mask3_fnmadd_ps (__m512, __m512, __m512, __mmask16);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFNMSUB132PD - Multiply Destination By Second Source, Negate, and Subtract First Source Float64 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W1 9E /r	vfnmsub132pd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Multiply float64 vector zmm1 and float64 vector S_{f64}(zmm3/m_t), negate, and subtract float64 vector zmm2 from the result, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between float64 vector zmm1 and the float64 vector result of the swizzle/broadcast/conversion process on memory or vector float64 zmm3, negates, and subtracts float64 vector zmm2. The final result is written into float64 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

x*y	z	RN/RU/RZ			RD		
+0	+0	(-0)	+ (-0)	= -0	(-0)	+ (-0)	= -0
+0	-0	(-0)	+ (+0)	= +0	(-0)	+ (+0)	= -0
-0	+0	(+0)	+ (-0)	= +0	(+0)	+ (-0)	= -0
-0	-0	(+0)	+ (+0)	= +0	(+0)	+ (+0)	= +0

Table 6.11: VFNMSUB outcome when adding zeros depending on rounding-mode

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = (-(zmm1[i+63:i] * tmpSrc3[i+63:i]) - zmm2[i+63:i])
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)? YES : NO`

Flush Tiny Results To Zero :
`(MXCSR.FZ)? YES : NO`

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_fnmsub_pd (__m512d, __m512d, __m512d);
__m512d    _mm512_mask_fnmsub_pd (__m512d, __mmask8, __m512d, __m512d);
__m512d    _mm512_mask3_fnmsub_pd (__m512d, __m512d, __m512d, __mmask8);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is

	in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFNMSUB132PS - Multiply Destination By Second Source, Negate, and Subtract First Source Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 9E /r	vfnmsub132ps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Multiply float32 vector zmm1 and float32 vector S_{f32}(zmm3/m_t), negate, and subtract float32 vector zmm2 from the result, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between float32 vector zmm1 and the float32 vector result of the swizzle/broadcast/conversion process on memory or vector float32 zmm3, negates, and subtracts float32 vector zmm2. The final result is written into float32 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

x*y	z	RN/RU/RZ			RD		
+0	+0	(-0)	+ (-0)	= -0	(-0)	+ (-0)	= -0
+0	-0	(-0)	+ (+0)	= +0	(-0)	+ (+0)	= -0
-0	+0	(+0)	+ (-0)	= +0	(+0)	+ (-0)	= -0
-0	-0	(+0)	+ (+0)	= +0	(+0)	+ (+0)	= +0

Table 6.12: VFNMSUB outcome when adding zeros depending on rounding-mode

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = (-(zmm1[i+31:i] * tmpSrc3[i+31:i]) - zmm2[i+31:i])
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
     (MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :

```
     (MXCSR.FZ)? YES : NO
```

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S_{f32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_fnmsub_ps (__m512, __m512, __m512);
__m512    _mm512_mask_fnmsub_ps (__m512, __mmask16, __m512, __m512);
__m512    _mm512_mask3_fnmsub_ps (__m512, __m512, __m512, __mmask16);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.

#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFNMSUB213PD - Multiply First Source By Destination, Negate, and Subtract Second Source Float64 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W1 AE /r	vfnmsub213pd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Multiply float64 vector zmm2 and float64 vector zmm1, negate, and subtract float64 vector S_{f64}(zmm3/m_t) from the result, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between float64 vector zmm2 and float64 vector zmm1, negates, and subtracts the float64 vector result of the swizzle/broadcast/conversion process on memory or vector float64 zmm3. The final sum is written into float64 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

x*y	z	RN/RU/RZ			RD		
+0	+0	(-0)	+ (-0)	= -0	(-0)	+ (-0)	= -0
+0	-0	(-0)	+ (+0)	= +0	(-0)	+ (+0)	= -0
-0	+0	(+0)	+ (-0)	= +0	(+0)	+ (-0)	= -0
-0	-0	(+0)	+ (+0)	= +0	(+0)	+ (+0)	= +0

Table 6.13: VFNMSUB outcome when adding zeros depending on rounding-mode

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = (-(zmm2[i+63:i] * zmm1[i+63:i]) - tmpSrc3[i+63:i])
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)?  YES : NO`

Flush Tiny Results To Zero :
`(MXCSR.FZ)?  YES : NO`

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S_{f64}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_fnmsub_pd (__m512d, __m512d, __m512d);
__m512d    _mm512_mask_fnmsub_pd (__m512d, __mmask8, __m512d, __m512d);
__m512d    _mm512_mask3_fnmsub_pd (__m512d, __m512d, __m512d, __mmask8);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is

	in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFNMSUB213PS - Multiply First Source By Destination, Negate, and Subtract Second Source Float32 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W0 AE /r	vfnmsub213ps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Multiply float32 vector zmm2 and float32 vector zmm1, negate, and subtract float32 vector S_{f32}(zmm3/m_t) from the result, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between float32 vector zmm2 and float32 vector zmm1, negates, and subtracts the float32 vector result of the swizzle/broadcast/conversion process on memory or vector float32 zmm3. The final sum is written into float32 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

x*y	z	RN/RU/RZ			RD		
+0	+0	(-0)	+ (-0)	= -0	(-0)	+ (-0)	= -0
+0	-0	(-0)	+ (+0)	= +0	(-0)	+ (+0)	= -0
-0	+0	(+0)	+ (-0)	= +0	(+0)	+ (-0)	= -0
-0	-0	(+0)	+ (+0)	= +0	(+0)	+ (+0)	= +0

Table 6.14: VFNMSUB outcome when adding zeros depending on rounding-mode

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = (-(zmm2[i+31:i] * zmm1[i+31:i]) - tmpSrc3[i+31:i])
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)? YES : NO`

Flush Tiny Results To Zero :
`(MXCSR.FZ)? YES : NO`

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512   _mm512_fnmsub_ps (__m512, __m512, __m512);
__m512   _mm512_mask_fnmsub_ps (__m512, __mmask16, __m512, __m512);
__m512   _mm512_mask3_fnmsub_ps (__m512, __m512, __m512, __mmask16);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.

#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFNMSUB231PD - Multiply First Source By Second Source, Negate, and Subtract Destination Float64 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W1 BE /r	vfnmsub231pd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Multiply float64 vector zmm2 and float64 vector S_{f64}(zmm3/m_t), negate, and subtract float64 vector zmm1 from the result, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between float64 vector zmm2 and the float64 vector result of the swizzle/broadcast/conversion process on memory or vector float64 zmm3, negates, and subtracts float64 vector zmm1. The final result is written into float64 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

x*y	z	RN/RU/RZ			RD		
+0	+0	(-0)	+ (-0)	= -0	(-0)	+ (-0)	= -0
+0	-0	(-0)	+ (+0)	= +0	(-0)	+ (+0)	= -0
-0	+0	(+0)	+ (-0)	= +0	(+0)	+ (-0)	= -0
-0	-0	(+0)	+ (+0)	= +0	(+0)	+ (+0)	= +0

Table 6.15: VFMADDN outcome when adding zeros depending on rounding-mode

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = (-(zmm2[i+63:i] * tmpSrc3[i+63:i]) - zmm1[i+63:i])
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
        (MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
        (MXCSR.FZ)?  YES : NO
```

Memory Up-conversion: S_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_fnmsub_pd (__m512d, __m512d, __m512d);
__m512d    _mm512_mask_fnmsub_pd (__m512d, __mmask8, __m512d, __m512d);
__m512d    _mm512_mask3_fnmsub_pd (__m512d, __m512d, __m512d, __mmask8);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is

	in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VFNMSUB231PS - Multiply First Source By Second Source, Negate, and Subtract Destination Float32 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W0 BE /r	vfnmsub231ps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Multiply float32 vector zmm2 and float32 vector S_{f32}(zmm3/m_t), negate, and subtract float32 vector zmm1 from the result, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between float32 vector zmm2 and the float32 vector result of the swizzle/broadcast/conversion process on memory or vector float32 zmm3, negates, and subtracts float32 vector zmm1. The final result is written into float32 vector zmm1.

Intermediate values are calculated to infinite precision, and are not truncated or rounded. All operations must be performed previous to final rounding.

x*y	z	RN/RU/RZ			RD		
+0	+0	(-0)	+ (-0)	= -0	(-0)	+ (-0)	= -0
+0	-0	(-0)	+ (+0)	= +0	(-0)	+ (+0)	= -0
-0	+0	(+0)	+ (-0)	= +0	(+0)	+ (-0)	= -0
-0	-0	(+0)	+ (+0)	= +0	(+0)	+ (+0)	= +0

Table 6.16: VFMADDN outcome when adding zeros depending on rounding-mode

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = (-(zmm2[i+31:i] * tmpSrc3[i+31:i]) - zmm1[i+31:i])
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)? YES : NO`

Flush Tiny Results To Zero :
`(MXCSR.FZ)? YES : NO`

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_fnmsub_ps (__m512, __m512, __m512);
__m512    _mm512_mask_fnmsub_ps (__m512, __mmask16, __m512, __m512);
__m512    _mm512_mask3_fnmsub_ps (__m512, __m512, __m512, __mmask16);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.

VGATHERDPD - Gather Float64 Vector With Signed Dword Indices

Opcode	Instruction	Description
MVEX.512.66.0F38.W1 92 /r /vsib	vgatherdpd zmm1 {k1}, $U_{f64}(mv_t)$	Gather float64 vector $U_{f64}(mv_t)$ into float64 vector zmm1 using doubleword indices and k1 as completion mask.

Description

A set of 8 memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$ with scale $SCALE$ are converted to a float64 vector. The result is written into float64 vector zmm1.

Note the special mask behavior as only a subset of the active elements of write mask k1 are actually operated on (as denoted by function $SELECT\_SUBSET$). There are only two guarantees about the function: (a) the destination mask is a subset of the source mask (identity is included), and (b) on a given invocation of the instruction, **at least** one element (the least significant enabled mask bit) will be selected from the source mask.

Programmers should always enforce the execution of a gather/scatter instruction to be re-executed (via a loop) until the full completion of the sequence (i.e. all elements of the gather/scatter sequence have been loaded/stored and hence, the write-mask bits all are zero).

Note that accessed element by will always access 64 bytes of memory. The memory region accessed by each element will always be between *elemen_linear_address & (∼0x3F)* and *(element_linear_address & (∼0x3F)) + 63* boundaries.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Note also the special mask behavior as the corresponding bits in write mask k1 are reset with each destination element being updated according to the subset of write mask k1. This is useful to allow conditional re-trigger of the instruction until all the elements from a given write mask have been successfully loaded.

The instruction will #GP fault if the destination vector zmm1 is the same as index vector $VINDEX$.

Operation

```
// instruction works over a subset of the write mask
ktemp = SELECT_SUBSET(k1)

// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 8; n++) {
  if (ktemp[n] != 0) {
```

```
      i = 64*n
      j = 32*n
      // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[j+31:j] * SCALE)
      pointer[63:0] = $mv_t$[n]
      zmm1[i+63:i] = UpConvLoad$_{f64}$(pointer)
      k1[n] = 0
    }
  }
  k1[15:8] = 0
```

SIMD Floating-Point Exceptions

None.

Memory Up-conversion: U$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_i32logather_pd (__m512i, void const*, int);
__m512d    _mm512_mask_i32logather_pd (__m512d, __mmask8, __m512i, void const*, int);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form, and corresponding write-mask bit is not zero.
#GP(0)	If a memory address is in a non-canonical form, and corresponding write-mask bit is not zero. If a memory operand linear address is not aligned to element-wise data granularity dictated by the UpConv and corresponding write-mask bit is not zero. If the destination vector is the same as the index vector [see .
#PF(fault-code)	If a memory operand linear address produces a page fault and corresponding write-mask bit is not zero.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If using a 16 bit effective address. If ModRM.rm is different than 100b. If no write mask is provided or selected write-mask is k0.

VGATHERDPS - Gather Float32 Vector With Signed Dword Indices

Opcode	Instruction	Description
MVEX.512.66.0F38.W0 92 /r /vsib	vgatherdps zmm1 {k1}, $U_{f32}(mv_t)$	Gather float32 vector $U_{f32}(mv_t)$ into float32 vector zmm1 using doubleword indices and k1 as completion mask.

Description

A set of 16 memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$ with scale $SCALE$ are converted to a float32 vector. The result is written into float32 vector zmm1.

Note the special mask behavior as only a subset of the active elements of write mask k1 are actually operated on (as denoted by function $SELECT\_SUBSET$). There are only two guarantees about the function: (a) the destination mask is a subset of the source mask (identity is included), and (b) on a given invocation of the instruction, **at least** one element (the least significant enabled mask bit) will be selected from the source mask.

Programmers should always enforce the execution of a gather/scatter instruction to be re-executed (via a loop) until the full completion of the sequence (i.e. all elements of the gather/scatter sequence have been loaded/stored and hence, the write-mask bits all are zero).

Note that accessed element by will always access 64 bytes of memory. The memory region accessed by each element will always be between *elemen_linear_address & (∼0x3F)* and *(element_linear_address & (∼0x3F)) + 63* boundaries.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Note also the special mask behavior as the corresponding bits in write mask k1 are reset with each destination element being updated according to the subset of write mask k1. This is useful to allow conditional re-trigger of the instruction until all the elements from a given write mask have been successfully loaded.

The instruction will #GP fault if the destination vector zmm1 is the same as index vector $VINDEX$.

Operation

```
// instruction works over a subset of the write mask
ktemp = SELECT_SUBSET(k1)

// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 16; n++) {
  if (ktemp[n] != 0) {
```

```
        i = 32*n
        // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[i+31:i] * SCALE)
        pointer[63:0] = $mv_t$[n]
        zmm1[i+31:i] = UpConvLoad$_{f32}$(pointer)
        k1[n] = 0
      }
    }
```

SIMD Floating-Point Exceptions

Invalid.

Memory Up-conversion: U_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float16 to float32	[rax] {float16}	2
100	uint8 to float32	[rax] {uint8}	1
101	sint8 to float32	[rax] {sint8}	1
110	uint16 to float32	[rax] {uint16}	2
111	sint16 to float32	[rax] {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

__m512	_mm512_i32gather_ps (__m512i, void const*, int);
__m512	_mm512_mask_i32gather_ps (__m512, __mmask16, __m512i, void const*, int);
__m512	_mm512_i32extgather_ps (__m512i, void const*, _MM_UPCONV_PS_ENUM, int, int);
__m512	_mm512_mask_i32extgather_ps (__m512, __mmask16, __m512i, void const*, _MM_UPCONV_PS_ENUM, int, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form, and corresponding write-mask bit is not zero.
#GP(0)	If a memory address is in a non-canonical form, and corresponding write-mask bit is not zero. If a memory operand linear address is not aligned to element-wise data granularity dictated by the UpConv and corresponding write-mask bit is not zero. If the destination vector is the same as the index vector [see .
#PF(fault-code)	If a memory operand linear address produces a page fault and corresponding write-mask bit is not zero.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If using a 16 bit effective address. If ModRM.rm is different than 100b. If no write mask is provided or selected write-mask is k0.

VGATHERPF0DPS - Gather Prefetch Float32 Vector With Signed Dword Indices Into L1

Opcode	**Instruction**	**Description**
MVEX.512.66.0F38.W0 C6 /1 /vsib	vgatherpf0dps $U_{f32}(mv_t)$ {k1}	Gather Prefetch float32 vector $U_{f32}(mv_t)$, using doubleword indices with T0 hint, under write-mask.

Description

A set of 16 float32 memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$ with scale $SCALE$ are prefetched from memory to L1 level of cache. If any memory access causes any type of memory exception, the memory access will be considered as completed (destination mask updated) and the exception ignored.

Note the special mask behavior as only a subset of the active elements of write mask k1 are actually operated on (as denoted by function $SELECT\_SUBSET$). There are only two guarantees about the function: (a) the destination mask is a subset of the source mask (identity is included), and (b) on a given invocation of the instruction, **at least** one element (the least significant enabled mask bit) will be selected from the source mask.

Programmers should always enforce the execution of a gather/scatter instruction to be re-executed (via a loop) until the full completion of the sequence (i.e. all elements of the prefetch sequence have been prefetched and hence, the write-mask bits all are zero).

Note that accessed element by will always access 64 bytes of memory. The memory region accessed by each element will always be between *elemen_linear_address & (∼0x3F)* and *(element_linear_address & (∼0x3F)) + 63* boundaries.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Note also the special mask behavior as the corresponding bits in write mask k1 are reset with each destination element being updated according to the subset of write mask k1. This is useful to allow conditional re-trigger of the instruction until all the elements from a given write mask have been successfully loaded.

Note that both gather and scatter prefetches set the access bit (A) in the related TLB page entry. Scatter prefetches (which prefetch data with RFO) do not set the dirty bit (D).

Operation

```
// instruction works over a subset of the write mask
ktemp = SELECT_SUBSET(k1)

exclusive = 0
evicthintpre = MVEX.EH
```

```
// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 16; n++) {
  if (ktemp[n] != 0) {
    i = 32*n
    // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[i+31:i] * SCALE)
    pointer[63:0] = $mv_t$[n]
    FetchL1cacheLine(pointer, exclusive, evicthintpre)
    k1[n] = 0
  }
}
```

SIMD Floating-Point Exceptions

None.

Memory Up-conversion: U$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float16 to float32	[rax] {float16}	2
100	uint8 to float32	[rax] {uint8}	1
101	sint8 to float32	[rax] {sint8}	1
110	uint16 to float32	[rax] {uint16}	2
111	sint16 to float32	[rax] {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

void	_mm512_prefetch_i32gather_ps (__m512i, void const*, int, int);
void	_mm512_mask_prefetch_i32gather_ps (__m512i, __mmask16, void const*, int, int);
void	_mm512_prefetch_i32extgather_ps (__m512i, void const*, _MM_UPCONV_PS_ENUM, int, int);
void	_mm512_mask_prefetch_i32extgather_ps (__m512i, __mmask16, void const*, _MM_UPCONV_PS_ENUM, int, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If using a 16 bit effective address. If ModRM.rm is different than 100b. If no write mask is provided or selected write-mask is k0.

VGATHERPF0HINTDPD - Gather Prefetch Float64 Vector Hint With Signed Dword Indices

Opcode	Instruction		Description
MVEX.512.66.0F38.W1 C6 /0 /vsib	vgatherpf0hintdpd {k1}	$U_{f64}(mv_t)$	Gather Prefetch float64 vector $U_{f64}(mv_t)$, using doubleword indices with T0 hint, under write-mask.

Description

The instruction specifies a set of 8 float64 memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$ with scale $SCALE$ as a performance hint that a real gather instruction with the same set of sources will be invoked. A programmer may execute this instruction before a real gather instruction to improve its performance.

This instruction is a hint and may be speculative, and may be dropped or specify invalid addresses without causing problems or memory related faults. This instructions does not modify any kind of architectural state (including the write-mask).

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Operation

```
// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 8; n++) {
  if (k1[n] != 0) {
    i = 64*n
    j = 32*n
    // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[j+31:j] * SCALE)
    pointer[63:0] = $mv_t$[n]
    HintPointer(pointer)
  }
}
```

SIMD Floating-Point Exceptions

None.

Memory Up-conversion: U_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

None

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If using a 16 bit effective address. If ModRM.rm is different than 100b. If no write mask is provided or selected write-mask is k0.

VGATHERPF0HINTDPS - Gather Prefetch Float32 Vector Hint With Signed Dword Indices

Opcode	Instruction		Description
MVEX.512.66.0F38.W0 C6 /0 /vsib	vgatherpf0hintdps {k1}	$U_{f32}(mv_t)$	Gather Prefetch float32 vector $U_{f32}(mv_t)$, using doubleword indices with T0 hint, under write-mask.

Description

The instruction specifies a set of 16 float32 memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$ with scale $SCALE$ as a performance hint that a real gather instruction with the same set of sources will be invoked. A programmer may execute this instruction before a real gather instruction to improve its performance.

This instruction is a hint and may be speculative, and may be dropped or specify invalid addresses without causing problems or memory related faults. This instructions does not modify any kind of architectural state (including the write-mask).

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Operation

```
// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[i+31:i] * SCALE)
    pointer[63:0] = $mv_t$[n]
    HintPointer(pointer)
  }
}
```

SIMD Floating-Point Exceptions

None.

Memory Up-conversion: U_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float16 to float32	[rax] {float16}	2
100	uint8 to float32	[rax] {uint8}	1
101	sint8 to float32	[rax] {sint8}	1
110	uint16 to float32	[rax] {uint16}	2
111	sint16 to float32	[rax] {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

None

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If using a 16 bit effective address. If ModRM.rm is different than 100b. If no write mask is provided or selected write-mask is k0.

VGATHERPF1DPS - Gather Prefetch Float32 Vector With Signed Dword Indices Into L2

Opcode	Instruction	Description
MVEX.512.66.0F38.W0 C6 /2 /vsib	vgatherpf1dps $U_{f32}(mv_t)$ {k1}	Gather Prefetch float32 vector $U_{f32}(mv_t)$, using doubleword indices with T1 hint, under write-mask.

Description

A set of 16 float32 memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$ with scale $SCALE$ are prefetched from memory to L2 level of cache. If any memory access causes any type of memory exception, the memory access will be considered as completed (destination mask updated) and the exception ignored.

Note the special mask behavior as only a subset of the active elements of write mask k1 are actually operated on (as denoted by function $SELECT\_SUBSET$). There are only two guarantees about the function: (a) the destination mask is a subset of the source mask (identity is included), and (b) on a given invocation of the instruction, **at least** one element (the least significant enabled mask bit) will be selected from the source mask.

Programmers should always enforce the execution of a gather/scatter instruction to be re-executed (via a loop) until the full completion of the sequence (i.e. all elements of the prefetch sequence have been prefetched and hence, the write-mask bits all are zero).

Note that accessed element by will always access 64 bytes of memory. The memory region accessed by each element will always be between *elemen_linear_address & (∼0x3F)* and *(element_linear_address & (∼0x3F)) + 63* boundaries.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Note also the special mask behavior as the corresponding bits in write mask k1 are reset with each destination element being updated according to the subset of write mask k1. This is useful to allow conditional re-trigger of the instruction until all the elements from a given write mask have been successfully loaded.

Note that both gather and scatter prefetches set the access bit (A) in the related TLB page entry. Scatter prefetches (which prefetch data with RFO) do not set the dirty bit (D).

Operation

```
// instruction works over a subset of the write mask
ktemp = SELECT_SUBSET(k1)

exclusive = 0
evicthintpre = MVEX.EH
```

```
// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 16; n++) {
  if (ktemp[n] != 0) {
    i = 32*n
    // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[i+31:i] * SCALE)
    pointer[63:0] = $mv_t$[n]
    FetchL2cacheLine(pointer, exclusive, evicthintpre)
    k1[n] = 0
  }
}
```

SIMD Floating-Point Exceptions

None.

Memory Up-conversion: U_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float16 to float32	[rax] {float16}	2
100	uint8 to float32	[rax] {uint8}	1
101	sint8 to float32	[rax] {sint8}	1
110	uint16 to float32	[rax] {uint16}	2
111	sint16 to float32	[rax] {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm512_prefetch_i32gather_ps (__m512i, void const*, int, int);

void _mm512_mask_prefetch_i32gather_ps (__m512i, __mmask16, void const*, int, int);

void _mm512_prefetch_i32extgather_ps (__m512i, void const*, _MM_UPCONV_PS_ENUM, int, int);

void _mm512_mask_prefetch_i32extgather_ps (__m512i, __mmask16, void const*, _MM_UPCONV_PS_ENUM, int, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If using a 16 bit effective address. If ModRM.rm is different than 100b. If no write mask is provided or selected write-mask is k0.

VGETEXPPD - Extract Float64 Vector of Exponents from Float64 Vector

Opcode	**Instruction**	**Description**
MVEX.512.66.0F38.W1 42 /r	vgetexppd zmm1 {k1}, S_{f64}(zmm2/m_t)	Extract float64 vector of exponents from vector S_{f64}(zmm2/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element exponent extraction from the Float64 vector result of the swizzle/broadcast/conversion process on memory or Float64 vector zmm2. The result is written into Float64 vector zmm1.

GetExp() returns the (un-biased) exponent n in floating-point format. That is, when $X = 1/16$, GetExp() returns the value -4, represented as C0800000 in IEEE single precision (for the single-precision version of the instruction). If the source is denormal, VGETEXP will normalize it prior to exponent extraction (unless DAZ=1).

GetExp() function follows Table 6.17 when dealing with floating-point special number.

Input	Result
NaN	quietized input NaN
$+\infty$	$+\infty$
+0	$-\infty$
-0	$-\infty$
$-\infty$	$+\infty$

Table 6.17: GetExp() special floating-point values behavior

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{f64}$(zmm2/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    zmm1[i+63:i] = GetExp(tmpSrc2[i+63:i])
```

```
    }
  }
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
        (MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
     Not Applicable
```

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d   _mm512_getexp_pd (__m512d);
__m512d   _mm512_mask_getexp_pd (__m512d, __mmask8, __m512d);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VGETEXPPS - Extract Float32 Vector of Exponents from Float32 Vector

Opcode	**Instruction**	**Description**
MVEX.512.66.0F38.W0 42 /r	vgetexpps zmm1 {k1}, S_{f32}(zmm2/m_t)	Extract float32 vector of exponents from vector S_{f32}(zmm2/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element exponent extraction from the Float32 vector result of the swizzle/broadcast/conversion process on memory or Float32 vector zmm2. The result is written into Float32 vector zmm1.

GetExp() returns the (un-biased) exponent n in floating-point format. That is, when $X = 1/16$, GetExp() returns the value -4, represented as C0800000 in IEEE single precision (for the single-precision version of the instruction). If the source is denormal, VGETEXP will normalize it prior to exponent extraction (unless DAZ=1).

GetExp() function follows Table 6.18 when dealing with floating-point special number.

Input	Result
NaN	quietized input NaN
$+\infty$	$+\infty$
+0	$-\infty$
-0	$-\infty$
$-\infty$	$+\infty$

Table 6.18: GetExp() special floating-point values behavior

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{f32}$(zmm2/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] = GetExp(tmpSrc2[i+31:i])
```

```
    }
}
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :

```
Not Applicable
```

Memory Up-conversion: S_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S_{f32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_getexp_ps (__m512);
__m512    _mm512_mask_getexp_ps (__m512, __mmask16, __m512);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VGETMANTPD - Extract Float64 Vector of Normalized Mantissas from Float64 Vector

Opcode	**Instruction**	**Description**
MVEX.512.66.0F3A.W1 26 /r ib	vgetmantpd zmm1 {k1}, S_{f64}(zmm2/m_t), $imm8$	Get Normalized Mantissa from float64 vector S_{f64}(zmm2/m_t) and store the result in zmm1, using $imm8$ for sign control and mantissa interval normalization, under write-mask.

Description

Performs an element-by-element conversion of the Float64 vector result of the swizzle/broadcast/conversion process on memory or Float64 vector zmm2 to Float64 values with the mantissa normalized to the interval specified by *interv* and sign dictated by the sign control parameter *sc*. The result is written into Float64 vector zmm1. Denormal values are explicitly normalized.

The formula for the operation is:

$GetMant(x) = \pm 2^k |x.significand|$

where:

$1 <= |x.significand| < 2$

Exponent k is dependent 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.

GetMant() function follows Table 6.19 when dealing with floating-point special numbers.

Input	Result	Exceptions/comments
NaN	QNaN(SRC)	Raises #I if sNaN
$+\infty$	$+\infty$	ignore $interv$
+0	+0.0	ignore $interv$
-0	(SC[0])? $+0.0 : -0.0$	ignore $interv$, set NaN/raise #I if SC[1]=1
$-\infty$	(SC[0])? $+\infty : -\infty$	ignore $interv$, set NaN/raise #I if SC[1]=1
< 0		set NaN/raise #I if SC[1]=1

Table 6.19: GetMant() special floating-point values behavior

This instruction is write-masked, 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.

Immediate Format

Normalization Interval	I_1	I_0
[1,2)	0	0
[1/2,2)	0	1
[1/2,1)	1	0
[3/4,3/2)	1	1

Sign Control	I_3	I_2
sign = sign(SRC)	0	0
sign = 0	0	1
DEST = NaN (#I) if sign(SRC) = 1	1	x

Operation

```
GetNormalizedMantissa(SRC , SignCtrl, Interv)
{
   // Extracting the SRC sign, exponent and mantissa fields
   SIGN  = (SignCtrl[0])? 0 : SRC[63];
   EXP   = SRC[63:52];
   FRACT = (DAZ && (EXP == 0))? 0 : SRC[51:0];

   // Check for NaN operand
   if(IsNaN(SRC)) {
      if(IsSNaN(SRC)) *set I flag*
      return QNaN(SRC)
   }

   // If SignCtrl[1] is set to 1, return NaN and set
   // exception flag if the operand is negative.
   // Note that -0.0 is included
   if( SignCtrl[1] && (SRC[63] == 1) )                          {
      *set I flag*
      return QNaN_Indefinite
   }

   // Check for +/-INF and +/-0
   if( ( EXP == 0x7FF && FRACTION == 0 )
    || ( EXP == 0 && FRACTION == 0 ) ) {
      DEST[63:0] =  (SIGN << 63) | (EXP[11:0] << 52) | FRACT[51:0];
      return DEST
   }

   // Normalize denormal operands
   //    note that denormal operands are treated as zero if
   //    DAZ is set to 1
   if((EXP == 0)  && (FRACTION !=0) {
       // JBIT is the hidden integral bit
      JBIT = 0;                      // Zero in case of denormal operands
       EXP = 03FFh;                    // Set exponent to BIAS
```

```
            While(JBIT == 0) {
               JBIT = FRACT[51];           // Obtain fraction MSB
               FRACT = FRACT << 1;         // Normalize mantissa
               EXP--;                      // and adjust exponent
            }
            *set D flag*
        }

        // Apply normalization intervals
        UNBIASED_EXP = EXP - 03FFh;        // get exponent in unbiased form
        IS_ODD_EXP   = UNBIASED_EXP[0];    // if the unbiased exponent odd?

        if( (Interv == 10b)
            || ( (Interv == 01b) && IS_ODD_EXP)
            || ( (Interv == 11b) && (FRACT[51]==1)) ) {
           EXP = 03FEh;                    // Set exponent to -1 (unbiased)
        }
        else {
           EXP = 03FFh;                    // Set exponent to 0 (unbiased)
        }

        // form the final destination
        DEST[63:0] =  (SIGN << 63) | (EXP[11:0] << 52) | FRACT[51:0];
        return DEST
     }


     sc = IMM8[3:2]
     interv = IMM8[1:0]

     if(source is a register operand and MVEX.EH bit is 1) {
        if(SSS[2]==1) Supress_Exception_Flags() // SAE
        tmpSrc2[511:0] = zmm2[511:0]
     } else {
        tmpSrc2[511:0] = SwizzUpConvLoad$_{f64}$(zmm2/$m_t$)
     }

     for (n = 0; n < 8; n++) {
       if(k1[n] != 0) {
         i = 64*n
         // float64 operation
         zmm1[i+63:i] = GetNormalizedMantissa(tmpSrc2[i+63:i], sc, interv)
       }
     }
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
        (MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
     Not Applicable
```

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

__m512d _mm512_getmant_pd (__m512d, _MM_MANTISSA_NORM_ENUM, _MM_MANTISSA_SIGN_ENUM);
__m512d _mm512_mask_getmant_pd (__m512d, __mmask8, __m512d, _MM_MANTISSA_NORM_ENUM, _MM_MANTISSA_SIGN_ENUM);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VGETMANTPS - Extract Float32 Vector of Normalized Mantissas from Float32 Vector

Opcode	Instruction	Description
MVEX.512.66.0F3A.W0 26 /r ib	vgetmantps zmm1 {k1}, S_{f32}(zmm2/m_t), $imm8$	Get Normalized Mantissa from float32 vector S_{f32}(zmm2/m_t) and store the result in zmm1, using $imm8$ for sign control and mantissa interval normalization, under write-mask.

Description

Performs an element-by-element conversion of the Float32 vector result of the swizzle/broadcast/conversion process on memory or Float32 vector zmm2 to Float32 values with the mantissa normalized to the interval specified by *interv* and sign dictated by the sign control parameter *sc*. The result is written into Float32 vector zmm1. Denormal values are explicitly normalized.

The formula for the operation is:

$GetMant(x) = \pm 2^k |x.significand|$

where:

$1 <= |x.significand| < 2$

Exponent k is dependent 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.

GetMant() function follows Table 6.20 when dealing with floating-point special numbers.

Input	Result	Exceptions/comments
NaN	QNaN(SRC)	Raises #I if sNaN
$+\infty$	$+\infty$	ignore $interv$
+0	+0.0	ignore $interv$
-0	(SC[0])? $+0.0 : -0.0$	ignore $interv$, set NaN/raise #I if SC[1]=1
$-\infty$	(SC[0])? $+\infty : -\infty$	ignore $interv$, set NaN/raise #I if SC[1]=1
< 0		set NaN/raise #I if SC[1]=1

Table 6.20: GetMant() special floating-point values behavior

This instruction is write-masked, 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.

Immediate Format

Normalization Interval	I_1	I_0
[1,2)	0	0
[1/2,2)	0	1
[1/2,1)	1	0
[3/4,3/2)	1	1

Sign Control	I_3	I_2
sign = sign(SRC)	0	0
sign = 0	0	1
DEST = NaN (#I) if sign(SRC) = 1	1	x

Operation

```
GetNormalizedMantissa(SRC , SignCtrl, Interv)
{
   // Extracting the SRC sign, exponent and mantissa fields
   SIGN  = (SignCtrl[0])? 0 : SRC[31];
   EXP   = SRC[30:23];
   FRACT = (DAZ && (EXP == 0))? 0 : SRC[22:0];

   // Check for NaN operand
   if(IsNaN(SRC)) {
      if(IsSNaN(SRC)) *set I flag*
      return QNaN(SRC)
   }

   // If SignCtrl[1] is set to 1, return NaN and set
   // exception flag if the operand is negative.
   // Note that -0.0 is included
   if( SignCtrl[1] && (SRC[31] == 1) )                          {
      *set I flag*
      return QNaN_Indefinite
   }

   // Check for +/-INF and +/-0
   if( ( EXP == 0xFF && FRACTION == 0 )
    || ( EXP == 0 && FRACTION == 0 ) ) {
      DEST[31:0] =  (SIGN << 31) | (EXP[7:0] << 23) | FRACT[22:0];
      return DEST
   }

   // Apply normalization intervals
   UNBIASED_EXP = EXP - 07Fh;         // get exponent in unbiased form
   IS_ODD_EXP   = UNBIASED_EXP[0];    // if the unbiased exponent odd?

   if( (Interv == 10b)
       || ( (Interv == 01b) && IS_ODD_EXP)
       || ( (Interv == 11b) && (FRACT[22]==1)) ) {
```

```
            EXP = 07Eh;                     // Set exponent to -1 (unbiased)
        }
        else {
            EXP = 07Fh;                     // Set exponent to 0 (unbiased)
        }

        // form the final destination
        DEST[31:0] =  (SIGN << 31) | (EXP[7:0] << 23) | FRACT[22:0];
        return DEST
    }


    sc = IMM8[3:2]
    interv = IMM8[1:0]

    if(source is a register operand and MVEX.EH bit is 1) {
        if(SSS[2]==1) Supress_Exception_Flags() // SAE
        tmpSrc2[511:0] = zmm2[511:0]
    } else {
        tmpSrc2[511:0] = SwizzUpConvLoad$_{f32}$(zmm2/$m_t$)
    }

    for (n = 0; n < 16; n++) {
      if(k1[n] != 0) {
        i = 32*n
        // float32 operation
        zmm1[i+31:i] = GetNormalizedMantissa(tmpSrc2[i+31:i], sc, interv)
      }
    }
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)? YES : NO`

Flush Tiny Results To Zero :
`Not Applicable`

Memory Up-conversion: S_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S_{f32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

__m512	_mm512_getmant_ps (__m512, _MM_MANTISSA_NORM_ENUM, _MM_MANTISSA_SIGN_ENUM);
__m512	_mm512_mask_getmant_ps (__m512, __mmask16, __m512, _MM_MANTISSA_NORM_ENUM, _MM_MANTISSA_SIGN_ENUM);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VGMAXABSPS - Absolute Maximum of Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 51 /r	vgmaxabsps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Determine the maximum of the absolute values of float32 vector zmm2 and float32 vector S_{f32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Determines the maximum of the absolute values of each pair of corresponding elements in float32 vector zmm2 and the float32 vector result of the swizzle/broadcast/conversion process on memory or float32 vector zmm3. The result is written into float32 vector zmm1.

Abs() returns the absolute value of one float32 argument. FpMax() returns the bigger of the two float32 arguments, following IEEE in general. NaN has special handling: If one source operand is NaN, then the other source operand is returned (choice made per-component). If both are NaN, then the unchanged NaN from the first source (here zmm2) is returned. Please note that if first source is a SNaN it won't be quietized, it will be returned without any modification. This differs from the new IEEE 754-08 rules, which states that in case of an input SNaN, its quietized version should be returned instead of the other value.

Another new IEEE 754-08 rule is that max(-0,+0) == max(+0,-0) == +0, which honors the sign, in contrast to the comparison rules for signed zero (stated above). D3D10.0 recommends the IEEE 754-08 behavior here, but it will not be enforced; it is permissible for the result of comparing zeros to be dependent on the order of parameters, using a comparison that ignores the signs.

This instruction treats input denormals as zeros according to the DAZ control bit, but it does not flush tiny results to zero.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
FpMaxAbs(A,B)
{
  if ((A == NaN) && (B == NaN))
    return Abs(A);
  else if (A == NaN)
    return Abs(B);
  else if (B == NaN)
```

```
    return Abs(A);
  else if ((Abs(A) == +inf) || (Abs(B) == +inf))
    return +inf;
  else if (Abs(A) >= Abs(B))
    return Abs(A);
  else
    return Abs(B);
}

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = FpMaxAbs(zmm2[i+31:i] , tmpSrc3[i+31:i])
  }
}
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)? YES : NO`

Flush Tiny Results To Zero :
`NO`

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_gmaxabs_ps (__m512, __m512);
__m512    _mm512_mask_gmaxabs_p s(__m512, __mmask16, __m512, __m512);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VGMAXPD - Maximum of Float64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W1 53 /r	vgmaxpd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Determine the maximum of float64 vector zmm2 and float64 vector S_{f64}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Determines the maximum value of each pair of corresponding elements in float64 vector zmm2 and the float64 vector result of the swizzle/broadcast/conversion process on memory or float64 vector zmm3. The result is written into float64 vector zmm1.

FpMax() returns the bigger of the two float32 arguments, following IEEE in general. NaN has special handling: If one source operand is NaN, then the other source operand is returned (choice made per-component). If both are NaN, then the unchanged NaN from the first source (here zmm2) is returned. Please note that if first source is a SNaN it won't be quietized, it will be returned without any modification. This differs from the new IEEE 754-08 rules, which states that in case of an input SNaN, its quietized version should be returned instead of the other value.

Another new IEEE 754-08 rule is that max(-0,+0) == max(+0,-0) == +0, which honors the sign, in contrast to the comparison rules for signed zero (stated above). D3D10.0 recommends the IEEE 754-08 behavior here, but it will not be enforced; it is permissible for the result of comparing zeros to be dependent on the order of parameters, using a comparison that ignores the signs.

This instruction treats input denormals as zeros according to the DAZ control bit, but it does not flush tiny results to zero.

The following table describes exception flags priority:

Input 1	Input 2	Flags	Comments
SNAN	denormal	#I	#I priority over #D
denormal	SNAN	#I	#I priority over #D
QNAN	denormal	none	QNaN rule priority over #D
denormal	QNAN	none	QNaN rule priority over #D
normal	denormal	#D	only if DAZ=0
denormal	normal	#D	only if DAZ=0
denormal	denormal	#D	only if DAZ=0

Table 6.21: Max exception flags priority

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
FpMax(A,B)
{
  if ((A == -0.0) && (B == +0.0)) return B;
  if ((A == +0.0) && (B == -0.0)) return A;
  if ((A == NaN) && (B == NaN))   return A;
  if (A == NaN)                   return B;
  if (B == NaN)                   return A;
  if (A == -inf)                  return B;
  if (B == -inf)                  return A;
  if (A == +inf)                  return A;
  if (B == +inf)                  return B;
  if (A >= B)                     return A;

  return B;
}

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad_f64(zmm3/m_t)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = FpMax(zmm2[i+63:i] , tmpSrc3[i+63:i])
  }
}
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)?  YES : NO`

Flush Tiny Results To Zero :
`NO`

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_gmax_pd (__m512d, __m512d);
__m512d    _mm512_mask_gmax_pd (__m512d, __mmask8,__m512d, __m512d);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VGMAXPS - Maximum of Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 53 /r	vgmaxps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Determine the maximum of float32 vector zmm2 and float32 vector S_{f32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Determines the maximum value of each pair of corresponding elements in float32 vector zmm2 and the float32 vector result of the swizzle/broadcast/conversion process on memory or float32 vector zmm3. The result is written into float32 vector zmm1.

FpMax() returns the bigger of the two float32 arguments, following IEEE in general. NaN has special handling: If one source operand is NaN, then the other source operand is returned (choice made per-component). If both are NaN, then the unchanged NaN from the first source (here zmm2) is returned. Please note that if first source is a SNaN it won't be quietized, it will be returned without any modification. This differs from the new IEEE 754-08 rules, which states that in case of an input SNaN, its quietized version should be returned instead of the other value.

Another new IEEE 754-08 rule is that max(-0,+0) == max(+0,-0) == +0, which honors the sign, in contrast to the comparison rules for signed zero (stated above). D3D10.0 recommends the IEEE 754-08 behavior here, but it will not be enforced; it is permissible for the result of comparing zeros to be dependent on the order of parameters, using a comparison that ignores the signs.

This instruction treats input denormals as zeros according to the DAZ control bit, but it does not flush tiny results to zero.

The following table describes exception flags priority:

Input 1	Input 2	Flags	Comments
SNAN	denormal	#I	#I priority over #D
denormal	SNAN	#I	#I priority over #D
QNAN	denormal	none	QNaN rule priority over #D
denormal	QNAN	none	QNaN rule priority over #D
normal	denormal	#D	only if DAZ=0
denormal	normal	#D	only if DAZ=0
denormal	denormal	#D	only if DAZ=0

Table 6.22: Max exception flags priority

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
FpMax(A,B)
{
  if ((A == -0.0) && (B == +0.0)) return B;
  if ((A == +0.0) && (B == -0.0)) return A;
  if ((A == NaN) && (B == NaN))   return A;
  if (A == NaN)                   return B;
  if (B == NaN)                   return A;
  if (A == -inf)                  return B;
  if (B == -inf)                  return A;
  if (A == +inf)                  return A;
  if (B == +inf)                  return B;
  if (A >= B)                     return A;

  return B;
}

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = FpMax(zmm2[i+31:i] , tmpSrc3[i+31:i])
  }
}
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)? YES : NO`

Flush Tiny Results To Zero :
`NO`

Memory Up-conversion: S_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S_{f32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_gmax_ps (__m512, __m512);
__m512    _mm512_mask_gmax_ps (__m512, __mmask16, __m512, __m512);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VGMINPD - Minimum of Float64 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W1 52 /r	vgminpd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Determine the minimum of float64 vector zmm2 and float64 vector S_{f64}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Determines the minimum value of each pair of corresponding elements in float64 vector zmm2 and the float64 vector result of the swizzle/broadcast/conversion process on memory or float64 vector zmm3. The result is written into float64 vector zmm1.

FpMin() returns the smaller of the two float32 arguments, following IEEE in general. NaN has special handling: If one source operand is NaN, then the other source operand is returned (choice made per-component). If both are NaN, then the unchanged NaN from the first source (here zmm2) is returned. Please note that if first source is a SNaN it won't be quietized, it will be returned without any modification. This differs from the new IEEE 754-08 rules, which states that in case of an input SNaN, its quietized version should be returned instead of the other value.

Another new IEEE 754-08 rule is that min(-0,+0) == min(+0,-0) == -0, which honors the sign, in contrast to the comparison rules for signed zero (stated above). D3D10.0 recommends the IEEE 754-08 behavior here, but it will not be enforced; it is permissible for the result of comparing zeros to be dependent on the order of parameters, using a comparison that ignores the signs.

This instruction treats input denormals as zeros according to the DAZ control bit, but it does not flush tiny results to zero.

The following table describes exception flags priority:

Input 1	Input 2	Flags	Comments
SNAN	denormal	#I	#I priority over #D
denormal	SNAN	#I	#I priority over #D
QNAN	denormal	none	QNaN rule priority over #D
denormal	QNAN	none	QNaN rule priority over #D
normal	denormal	#D	only if DAZ=0
denormal	normal	#D	only if DAZ=0
denormal	denormal	#D	only if DAZ=0

Table 6.23: Min exception flags priority

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
FpMin(A,B)
{
  if ((A == -0.0) && (B == +0.0)) return A;
  if ((A == +0.0) && (B == -0.0)) return B;
  if ((A == NaN) && (B == NaN))   return A;
  if (A == NaN)                   return B;
  if (B == NaN)                   return A;
  if (A == -inf)                  return A;
  if (B == -inf)                  return B;
  if (A == +inf)                  return B;
  if (B == +inf)                  return A;
  if (A < B)                      return A;

  return B;
}

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = FpMin(zmm2[i+63:i] , tmpSrc3[i+63:i])
  }
}
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)?  YES : NO`

Flush Tiny Results To Zero :
`NO`

Memory Up-conversion: S_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S_{f64}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_gmin_pd (__m512d, __m512d);
__m512d    _mm512_mask_gmin_pd (__m512d, __mmask8, __m512d, __m512d);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VGMINPS - Minimum of Float32 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W0 52 /r	vgminps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Determine the minimum of float32 vector zmm2 and float32 vector S_{f32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Determines the minimum value of each pair of corresponding elements in float32 vector zmm2 and the float32 vector result of the swizzle/broadcast/conversion process on memory or float32 vector zmm3. The result is written into float32 vector zmm1.

FpMin() returns the smaller of the two float32 arguments, following IEEE in general. NaN has special handling: If one source operand is NaN, then the other source operand is returned (choice made per-component). If both are NaN, then the unchanged NaN from the first source (here zmm2) is returned. Please note that if first source is a SNaN it won't be quietized, it will be returned without any modification. This differs from the new IEEE 754-08 rules, which states that in case of an input SNaN, its quietized version should be returned instead of the other value.

Another new IEEE 754-08 rule is that min(-0,+0) == min(+0,-0) == -0, which honors the sign, in contrast to the comparison rules for signed zero (stated above). D3D10.0 recommends the IEEE 754-08 behavior here, but it will not be enforced; it is permissible for the result of comparing zeros to be dependent on the order of parameters, using a comparison that ignores the signs.

This instruction treats input denormals as zeros according to the DAZ control bit, but it does not flush tiny results to zero.

The following table describes exception flags priority:

Input 1	Input 2	Flags	Comments
SNAN	denormal	#I	#I priority over #D
denormal	SNAN	#I	#I priority over #D
QNAN	denormal	none	QNaN rule priority over #D
denormal	QNAN	none	QNaN rule priority over #D
normal	denormal	#D	only if DAZ=0
denormal	normal	#D	only if DAZ=0
denormal	denormal	#D	only if DAZ=0

Table 6.24: Min exception flags priority

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
FpMin(A,B)
{
  if ((A == -0.0) && (B == +0.0)) return A;
  if ((A == +0.0) && (B == -0.0)) return B;
  if ((A == NaN) && (B == NaN))   return A;
  if (A == NaN)                   return B;
  if (B == NaN)                   return A;
  if (A == -inf)                  return A;
  if (B == -inf)                  return B;
  if (A == +inf)                  return B;
  if (B == +inf)                  return A;
  if (A < B)                      return A;

  return B;
}

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = FpMin(zmm2[i+31:i] , tmpSrc3[i+31:i])
  }
}
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)? YES : NO`

Flush Tiny Results To Zero :
`NO`

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_gmin_ps(__m512, __m512);
__m512    _mm512_mask_gmin_ps(__m512, __mmask16, __m512, __m512);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VLOADUNPACKHD - Load Unaligned High And Unpack To Doubleword Vector

Opcode	Instruction	Description
MVEX.512.0F38.W0 D4 /r	vloadunpackhd zmm1 {k1}, $U_{i32}(m_t)$	Load high 64-byte-aligned portion of unaligned doubleword stream $U_{i32}(m_t$ - 64), unpack mask-enabled elements that fall in that portion, and store those elements in doubleword vector zmm1, under write-mask.

Description

The high-64-byte portion of the byte/word/doubleword stream starting at the element-aligned address $(m_t - 64)$ is loaded, converted and expanded into the write-mask-enabled elements of doubleword vector zmm1. The number of set bits in the write-mask determines the length of the converted doubleword stream, as each converted doubleword is mapped to exactly one of the doubleword elements in zmm1, skipping over write-masked elements of zmm1.

This instruction only transfers those converted doublewords (if any) in the stream that occur at or after the first 64-byte-aligned address following $(m_t - 64)$ (that is, in the high cache line of the memory stream for the current implementation). Elements in zmm1 that don't map to those stream doublewords are left unchanged. The vloadunpackld instruction is used to load the part of the stream before the first 64-byte-aligned address preceding m_t.

In conjunction with vloadunpackld, this instruction is useful for re-expanding data that was packed into a queue. Also in conjunction with vloadunpackld, it allows unaligned vector loads (that is, vector loads that are only element-wise, not vector-wise, aligned); use a mask of 0xFFFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector load would be:

```
// assume memory location is pointed by register rax
vloadunpackld  v0 {k1}, [rax]
vloadunpackhd  v0 {k1}, [rax+64]
```

This instruction does not have broadcast support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Note that this instruction will always access 64 bytes of memory. The memory region accessed will always be between *linear_address & (∼0x3F)* and *(linear_address & (∼0x3F)) + 63* boundaries.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand. The instruction will not produce any #GP or

#SS fault due to address canonicity nor #PF fault if the address is aligned to a 64-byte boundary. Additionally, A/D bits in the page table will not be updated.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are modified in zmm1. Elements in zmm1 with the corresponding bit clear in vector mask1 retain their previous values. However, see above for unusual aspects of the write-mask's operation with this instruction.

Operation

```
loadOffset = 0
upSize = UpConvLoadSizeOf$_{i32}$(SSS[2:0])
foundNext64BytesBoundary = false

pointer = $m_t$ - 64
for (n = 0; n < 16; n++) {
   if(k1[n] != 0) {
      if (foundNext64BytesBoundary == false) {
         if ( ( ((pointer + (loadOffset+1)*upSize ) % 64) == 0 ) {
            foundNext64BytesBoundary = true
         }
      } else {
         i = 32*n
         zmm1[i+31:i] = UpConvLoad$_{i32}$(pointer + upSize*loadOffset)
      }
      loadOffset++
   }
}
```

Flags Affected

None.

Memory Up-conversion: U$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	1
101	sint8 to sint32	[rax] {sint8}	1
110	uint16 to uint32	[rax] {uint16}	2
111	sint16 to sint32	[rax] {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i   _mm512_extloadunpackhi_epi32 (__m512i, void const*,
          _MM_UPCONV_EPI32_ENUM, int);
__m512i   _mm512_mask_extloadunpackhi_epi32 (__m512i, __mmask16, void const*,
          _MM_UPCONV_EPI32_ENUM, int);
__m512i   _mm512_loadunpackhi_epi32 (__m512i, void const*);
__m512i   _mm512_mask_loadunpackhi_epi32 (__m512i, __mmask16, void const*);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the UpConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the second operand is not a memory location.

VLOADUNPACKHPD - Load Unaligned High And Unpack To Float64 Vector

Opcode	Instruction	Description
MVEX.512.0F38.W1 D5 /r	vloadunpackhpd zmm1 {k1}, $U_{f64}(m_t)$	Load high 64-byte-aligned portion of unaligned float64 stream $U_{f64}(m_t$ - 64), unpack mask-enabled elements that fall in that portion, and store those elements in float64 vector zmm1, under write-mask.

Description

The high-64-byte portion of the quadword stream starting at the element-aligned address $(m_t - 64)$ is loaded, converted and expanded into the write-mask-enabled elements of quadword vector zmm1. The number of set bits in the write-mask determines the length of the converted quadword stream, as each converted quadword is mapped to exactly one of the quadword elements in zmm1, skipping over write-masked elements of zmm1.

This instruction only transfers those converted quadwords (if any) in the stream that occur at or after the first 64-byte-aligned address following $(m_t - 64)$ (that is, in the high cache line of the memory stream for the current implementation). Elements in zmm1 that don't map to those stream quadwords are left unchanged. The vloadunpacklpd instruction is used to load the part of the stream before the first 64-byte-aligned address preceding m_t.

In conjunction with vloadunpacklpd, this instruction is useful for re-expanding data that was packed into a queue. Also in conjunction with vloadunpacklpd, it allows unaligned vector loads (that is, vector loads that are only element-wise, not vector-wise, aligned); use a mask of 0xFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector load would be:

```
// assume memory location is pointed by register rax
vloadunpacklpd  v0 {k1}, [rax]
vloadunpackhpd  v0 {k1}, [rax+64]
```

This instruction does not have broadcast support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Note that this instruction will always access 64 bytes of memory. The memory region accessed will always be between *linear_address & (∼0x3F)* and *(linear_address & (∼0x3F)) + 63* boundaries.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand. The instruction will not produce any #GP or #SS fault due to address canonicity nor #PF fault if the address is aligned to a 64-byte boundary. Additionally, A/D bits in the page table will not be updated.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are modified in zmm1. Elements in zmm1 with the corresponding bit clear in vector mask1 retain their previous values. However, see above for unusual aspects of the write-mask's operation with this instruction.

Operation

```
loadOffset = 0
upSize = UpConvLoadSizeOf$_{f64}$(SSS[2:0])
foundNext64BytesBoundary = false

pointer = $m_t$ - 64
for (n = 0; n < 8; n++) {
   if(k1[n] != 0) {
      if (foundNext64BytesBoundary == false) {
         if ( ( ((pointer + (loadOffset+1)*upSize ) % 64) == 0 ) {
            foundNext64BytesBoundary = true
         }
      } else {
         i = 64*n
         zmm1[i+63:i] = UpConvLoad$_{f64}$(pointer + upSize*loadOffset)
      }
      loadOffset++
   }
}
```

SIMD Floating-Point Exceptions

None.

Memory Up-conversion: U$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

__m512d _mm512_extloadunpackhi_pd (__m512d, void const*, _MM_UPCONV_PD_ENUM, int);
__m512d _mm512_mask_extloadunpackhi_pd (__m512d, __mmask8, void const*, _MM_UPCONV_PD_ENUM, int);
__m512d _mm512_loadunpackhi_pd (__m512d, void const*);
__m512d _mm512_mask_loadunpackhi_pd (__m512d, __mmask8, void const*);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the UpConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the second operand is not a memory location.

VLOADUNPACKHPS - Load Unaligned High And Unpack To Float32 Vector

Opcode	Instruction	Description
MVEX.512.0F38.W0 D5 /r	vloadunpackhps zmm1 {k1}, $U_{f32}(m_t)$	Load high 64-byte-aligned portion of unaligned float32 stream $U_{f32}(m_t$ - 64), unpack mask-enabled elements that fall in that portion, and store those elements in float32 vector zmm1, under write-mask.

Description

The high-64-byte portion of the byte/word/doubleword stream starting at the element-aligned address $(m_t - 64)$ is loaded, converted and expanded into the write-mask-enabled elements of doubleword vector zmm1. The number of set bits in the write-mask determines the length of the converted doubleword stream, as each converted doubleword is mapped to exactly one of the doubleword elements in zmm1, skipping over write-masked elements of zmm1.

This instruction only transfers those converted doublewords (if any) in the stream that occur at or after the first 64-byte-aligned address following $(m_t - 64)$ (that is, in the high cache line of the memory stream for the current implementation). Elements in zmm1 that don't map to those stream doublewords are left unchanged. The vloadunpacklps instruction is used to load the part of the stream before the first 64-byte-aligned address preceding m_t.

In conjunction with vloadunpacklps, this instruction is useful for re-expanding data that was packed into a queue. Also in conjunction with vloadunpacklps, it allows unaligned vector loads (that is, vector loads that are only element-wise, not vector-wise, aligned); use a mask of 0xFFFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector load would be:

```
// assume memory location is pointed by register rax
vloadunpacklps  v0 {k1}, [rax]
vloadunpackhps  v0 {k1}, [rax+64]
```

This instruction does not have broadcast support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Note that this instruction will always access 64 bytes of memory. The memory region accessed will always be between *linear_address & (∼0x3F)* and *(linear_address & (∼0x3F)) + 63* boundaries.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand. The instruction will not produce any #GP or

#SS fault due to address canonicity nor #PF fault if the address is aligned to a 64-byte boundary. Additionally, A/D bits in the page table will not be updated.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are modified in zmm1. Elements in zmm1 with the corresponding bit clear in vector mask1 retain their previous values. However, see above for unusual aspects of the write-mask's operation with this instruction.

Operation

```
loadOffset = 0
upSize = UpConvLoadSizeOf$_{f32}$(SSS[2:0])
foundNext64BytesBoundary = false

pointer = $m_t$ - 64
for (n = 0; n < 16; n++) {
   if(k1[n] != 0) {
      if (foundNext64BytesBoundary == false) {
         if ( ( ((pointer + (loadOffset+1)*upSize ) % 64) == 0 ) {
            foundNext64BytesBoundary = true
         }
      } else {
         i = 32*n
         zmm1[i+31:i] = UpConvLoad$_{f32}$(pointer + upSize*loadOffset)
      }
      loadOffset++
   }
}
```

SIMD Floating-Point Exceptions

Invalid.

Memory Up-conversion: U$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float16 to float32	[rax] {float16}	2
100	uint8 to float32	[rax] {uint8}	1
101	sint8 to float32	[rax] {sint8}	1
110	uint16 to float32	[rax] {uint16}	2
111	sint16 to float32	[rax] {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

__m512	_mm512_extloadunpackhi_ps (__m512, void const*, _MM_UPCONV_PS_ENUM, int);
__m512	_mm512_mask_extloadunpackhi_ps (__m512, __mmask16, void const*, _MM_UPCONV_PS_ENUM, int);
__m512	_mm512_loadunpackhi_ps (__m512, void const*);
__m512	_mm512_mask_loadunpackhi_ps (__m512, __mmask16, void const*);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the UpConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the second operand is not a memory location.

VLOADUNPACKHQ - Load Unaligned High And Unpack To Int64 Vector

Opcode	Instruction	Description
MVEX.512.0F38.W1 D4 /r	vloadunpackhq zmm1 {k1}, $U_{i64}(m_t)$	Load high 64-byte-aligned portion of unaligned int64 stream $U_{i64}(m_t$ - 64), unpack mask-enabled elements that fall in that portion, and store those elements in int64 vector zmm1, under write-mask.

Description

The high-64-byte portion of the quadword stream starting at the element-aligned address $(m_t - 64)$ is loaded, converted and expanded into the write-mask-enabled elements of quadword vector zmm1. The number of set bits in the write-mask determines the length of the converted quadword stream, as each converted quadword is mapped to exactly one of the quadword elements in zmm1, skipping over write-masked elements of zmm1.

This instruction only transfers those converted quadwords (if any) in the stream that occur at or after the first 64-byte-aligned address following $(m_t - 64)$ (that is, in the high cache line of the memory stream for the current implementation). Elements in zmm1 that don't map to those stream quadwords are left unchanged. The vloadunpacklq instruction is used to load the part of the stream before the first 64-byte-aligned address preceding m_t.

In conjunction with vloadunpacklq, this instruction is useful for re-expanding data that was packed into a queue. Also in conjunction with vloadunpacklq, it allows unaligned vector loads (that is, vector loads that are only element-wise, not vector-wise, aligned); use a mask of 0xFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector load would be:

```
// assume memory location is pointed by register rax
vloadunpacklq  v0 {k1}, [rax]
vloadunpackhq  v0 {k1}, [rax+64]
```

This instruction does not have broadcast support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Note that this instruction will always access 64 bytes of memory. The memory region accessed will always be between *linear_address & (∼0x3F)* and *(linear_address & (∼0x3F)) + 63* boundaries.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand. The instruction will not produce any #GP or #SS fault due to address canonicity nor #PF fault if the address is aligned to a 64-byte boundary. Additionally, A/D bits in the page table will not be updated.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are modified in zmm1. Elements in zmm1 with the corresponding bit clear in vector mask1 retain their previous values. However, see above for unusual aspects of the write-mask's operation with this instruction.

Operation

```
loadOffset = 0
upSize = UpConvLoadSizeOf$_{i64}$(SSS[2:0])
foundNext64BytesBoundary = false

pointer = $m_t$ - 64
for (n = 0; n < 8; n++) {
   if(k1[n] != 0) {
      if (foundNext64BytesBoundary == false) {
         if ( ( ((pointer + (loadOffset+1)*upSize ) % 64) == 0 ) {
            foundNext64BytesBoundary = true
         }
      } else {
         i = 64*n
         zmm1[i+63:i] = UpConvLoad$_{i64}$(pointer + upSize*loadOffset)
      }
      loadOffset++
   }
}
```

Flags Affected

None.

Memory Up-conversion: U$_{i64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i   _mm512_extloadunpackhi_epi64 (__m512i, void const*, _MM_UPCONV_EPI64_ENUM, int);
__m512i   _mm512_mask_extloadunpackhi_epi64 (__m512i, __mmask8, void const*, _MM_UPCONV_EPI64_ENUM, int);
__m512i   _mm512_loadunpackhi_epi64 (__m512i, void const*);
__m512i   _mm512_mask_loadunpackhi_epi64 (__m512i, __mmask8, void const*);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the UpConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the second operand is not a memory location.

VLOADUNPACKLD - Load Unaligned Low And Unpack To Doubleword Vector

Opcode	Instruction	Description
MVEX.512.0F38.W0 D0 /r	vloadunpackld zmm1 {k1}, $U_{i32}(m_t)$	Load low 64-byte-aligned portion of unaligned doubleword stream $U_{i32}(m_t)$, unpack mask-enabled elements that fall in that portion, and store those elements in doubleword vector zmm1, under write-mask.

Description

The low-64-byte portion of the byte/word/doubleword stream starting at the element-aligned address m_t is loaded, converted and expanded into the write-mask-enabled elements of doubleword vector zmm1. The number of set bits in the write-mask determines the length of the converted doubleword stream, as each converted doubleword is mapped to exactly one of the doubleword elements in zmm1, skipping over write-masked elements of zmm1.

This instruction only transfers those converted doublewords (if any) in the stream that occur before the first 64-byte-aligned address following m_t (that is, in the low cache line of the memory stream in the current implementation). Elements in zmm1 that don't map to those converted stream doublewords are left unchanged. The vloadunpackhd instruction is used to load the part of the stream at or after the first 64-byte-aligned address preceding m_t.

In conjunction with vloadunpackhd, this instruction is useful for re-expanding data that was packed into a queue. Also in conjunction with vloadunpackhd, it allows unaligned vector loads (that is, vector loads that are only element-wise, not vector-wise, aligned); use a mask of 0xFFFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector load would be:

```
// assume memory location is pointed by register rax
vloadunpackld  v0 {k1}, [rax]
vloadunpackhd  v0 {k1}, [rax+64]
```

This instruction does not have broadcast support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Note that this instruction will always access 64 bytes of memory. The memory region accessed will always be between *linear_address & (∼0x3F)* and *(linear_address & (∼0x3F)) + 63* boundaries.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are modified in zmm1. Elements in zmm1 with the corresponding bit clear in vector mask1 retain their previous values. However, see above for unusual aspects of the write-mask's operation with this instruction.

Operation

```
loadOffset = 0
upSize = UpConvLoadSizeOf$_{i32}$(SSS[2:0])

for(n = 0 ;n < 16; n++) {
  i = 32*n
  if (k1[n] != 0) {
    zmm1[i+31:i] = UpConvLoad$_{i32}$($m_t$+upSize*loadOffset)
    loadOffset++
    if ( ( ($m_t$ + upSize*loadOffset) % 64) == 0) {
      break
    }
  }
}
```

Flags Affected

None.

Memory Up-conversion: U$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	1
101	sint8 to sint32	[rax] {sint8}	1
110	uint16 to uint32	[rax] {uint16}	2
111	sint16 to sint32	[rax] {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

__m512i	_mm512_extloadunpacklo_epi32 (__m512i, void const*, _MM_UPCONV_EPI32_ENUM, int);
__m512i	_mm512_mask_extloadunpacklo_epi32 (__m512i, __mmask16, void const*, _MM_UPCONV_EPI32_ENUM, int);
__m512i	_mm512_loadunpacklo_epi32 (__m512i, void const*);
__m512i	_mm512_mask_loadunpacklo_epi32 (__m512i, __mmask16, void const*);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the UpConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the second operand is not a memory location.

VLOADUNPACKLPD - Load Unaligned Low And Unpack To Float64 Vector

Opcode	Instruction	Description
MVEX.512.0F38.W1 D1 /r	vloadunpacklpd zmm1 {k1}, $U_{f64}(m_t)$	Load low 64-byte-aligned portion of unaligned float64 stream $U_{f64}(m_t)$, unpack mask-enabled elements that fall in that portion, and store those elements in float64 vector zmm1, under write-mask.

Description

The low-64-byte portion of the quadword stream starting at the element-aligned address m_t is loaded, converted and expanded into the write-mask-enabled elements of quadword vector zmm1. The number of set bits in the write-mask determines the length of the converted quadword stream, as each converted quadword is mapped to exactly one of the quadword elements in zmm1, skipping over write-masked elements of zmm1.

This instruction only transfers those converted quadwords (if any) in the stream that occur before the first 64-byte-aligned address following m_t (that is, in the low cache line of the memory stream in the current implementation). Elements in zmm1 that don't map to those converted stream quadwords are left unchanged. The vloadunpackhq instruction is used to load the part of the stream at or after the first 64-byte-aligned address preceding m_t.

In conjunction with vloadunpackhpd, this instruction is useful for re-expanding data that was packed into a queue. Also in conjunction with vloadunpackhpd, it allows unaligned vector loads (that is, vector loads that are only element-wise, not vector-wise, aligned); use a mask of 0xFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector load would be:

```
// assume memory location is pointed by register rax
vloadunpacklpd  v0 {k1}, [rax]
vloadunpackhpd  v0 {k1}, [rax+64]
```

This instruction does not have broadcast support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Note that this instruction will always access 64 bytes of memory. The memory region accessed will always be between *linear_address & (∼0x3F)* and *(linear_address & (∼0x3F)) + 63* boundaries.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are modified in zmm1. Elements in zmm1 with the corresponding bit clear in vector mask1 retain their previous values. However, see above for unusual aspects of the write-mask's operation with this instruction.

Operation

```
loadOffset = 0
upSize = UpConvLoadSizeOf$_{f64}$(SSS[2:0])

for(n = 0 ;n < 8; n++) {
  i = 64*n
  if (k1[n] != 0) {
    zmm1[i+63:i] = UpConvLoad$_{f64}$($m_t$+upSize*loadOffset)
    loadOffset++
    if ( ( ($m_t$ + upSize*loadOffset) % 64) == 0) {
      break
    }
  }
}
```

SIMD Floating-Point Exceptions

None.

Memory Up-conversion: U$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

__m512d	_mm512_extloadunpacklo_pd (__m512d, void const*, _MM_UPCONV_PD_ENUM, int);
__m512d	_mm512_mask_extloadunpacklo_pd (__m512d, __mmask8, void const*, _MM_UPCONV_PD_ENUM, int);
__m512d	_mm512_loadunpacklo_pd (__m512d, void const*);
__m512d	_mm512_mask_loadunpacklo_pd (__m512d, __mmask8, void const*);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the UpConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the second operand is not a memory location.

VLOADUNPACKLPS - Load Unaligned Low And Unpack To Float32 Vector

Opcode	Instruction	Description
MVEX.512.0F38.W0 D1 /r	vloadunpacklps zmm1 {k1}, $U_{f32}(m_t)$	Load low 64-byte-aligned portion of unaligned float32 stream $U_{f32}(m_t)$, unpack mask-enabled elements that fall in that portion, and store those elements in float32 vector zmm1, under write-mask.

Description

The low-64-byte portion of the byte/word/doubleword stream starting at the element-aligned address m_t is loaded, converted and expanded into the write-mask-enabled elements of doubleword vector zmm1. The number of set bits in the write-mask determines the length of the converted doubleword stream, as each converted doubleword is mapped to exactly one of the doubleword elements in zmm1, skipping over write-masked elements of zmm1.

This instruction only transfers those converted doublewords (if any) in the stream that occur before the first 64-byte-aligned address following m_t (that is, in the low cache line of the memory stream in the current implementation). Elements in zmm1 that don't map to those converted stream doublewords are left unchanged. The vloadunpackhd instruction is used to load the part of the stream at or after the first 64-byte-aligned address preceding m_t.

In conjunction with vloadunpackhps, this instruction is useful for re-expanding data that was packed into a queue. Also in conjunction with vloadunpackhps, it allows unaligned vector loads (that is, vector loads that are only element-wise, not vector-wise, aligned); use a mask of 0xFFFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector load would be:

```
// assume memory location is pointed by register rax
vloadunpacklps  v0 {k1}, [rax]
vloadunpackhps  v0 {k1}, [rax+64]
```

This instruction does not have broadcast support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Note that this instruction will always access 64 bytes of memory. The memory region accessed will always be between *linear_address & (∼0x3F)* and *(linear_address & (∼0x3F)) + 63* boundaries.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are modified in zmm1. Elements in zmm1 with the corresponding bit clear in vector mask1 retain their previous values. However, see above for unusual aspects of the write-mask's operation with this instruction.

Operation

```
loadOffset = 0
upSize = UpConvLoadSizeOf$_{f32}$(SSS[2:0])

for(n = 0 ;n < 16; n++) {
  i = 32*n
  if (k1[n] != 0) {
    zmm1[i+31:i] = UpConvLoad$_{f32}$($m_t$+upSize*loadOffset)
    loadOffset++
    if ( ( ($m_t$ + upSize*loadOffset) % 64) == 0) {
      break
    }
  }
}
```

SIMD Floating-Point Exceptions

Invalid.

Memory Up-conversion: U$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float16 to float32	[rax] {float16}	2
100	uint8 to float32	[rax] {uint8}	1
101	sint8 to float32	[rax] {sint8}	1
110	uint16 to float32	[rax] {uint16}	2
111	sint16 to float32	[rax] {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

__m512	_mm512_extloadunpacklo_ps (__m512, void const*, _MM_UPCONV_PS_ENUM, int);
__m512	_mm512_mask_extloadunpacklo_ps (__m512, __mmask16, void const*, _MM_UPCONV_PS_ENUM, int);
__m512	_mm512_loadunpacklo_ps (__m512, void const*);
__m512	_mm512_mask_loadunpacklo_ps (__m512, __mmask16, void const*);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the UpConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the second operand is not a memory location.

VLOADUNPACKLQ - Load Unaligned Low And Unpack To Int64 Vector

Opcode	Instruction	Description
MVEX.512.0F38.W1 D0 /r	vloadunpacklq zmm1 {k1}, $U_{i64}(m_t)$	Load low 64-byte-aligned portion of unaligned int64 stream $U_{i64}(m_t)$, unpack mask-enabled elements that fall in that portion, and store those elements in int64 vector zmm1, under write-mask.

Description

The low-64-byte portion of the quadword stream starting at the element-aligned address m_t is loaded, converted and expanded into the write-mask-enabled elements of quadword vector zmm1. The number of set bits in the write-mask determines the length of the converted quadword stream, as each converted quadword is mapped to exactly one of the quadword elements in zmm1, skipping over write-masked elements of zmm1.

This instruction only transfers those converted quadwords (if any) in the stream that occur before the first 64-byte-aligned address following m_t (that is, in the low cache line of the memory stream in the current implementation). Elements in zmm1 that don't map to those converted stream quadwords are left unchanged. The vloadunpackhq instruction is used to load the part of the stream at or after the first 64-byte-aligned address preceding m_t.

In conjunction with vloadunpackhq, this instruction is useful for re-expanding data that was packed into a queue. Also in conjunction with vloadunpackhq, it allows unaligned vector loads (that is, vector loads that are only element-wise, not vector-wise, aligned); use a mask of 0xFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector load would be:

```
// assume memory location is pointed by register rax
vloadunpacklq  v0 {k1}, [rax]
vloadunpackhq  v0 {k1}, [rax+64]
```

This instruction does not have broadcast support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Note that this instruction will always access 64 bytes of memory. The memory region accessed will always be between *linear_address & (∼0x3F)* and *(linear_address & (∼0x3F)) + 63* boundaries.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are modified in zmm1. Elements in zmm1 with the corresponding bit clear in vector mask1 retain their previous values. However, see above for unusual aspects of the write-mask's operation with this instruction.

Operation

```
loadOffset = 0
upSize = UpConvLoadSizeOf$_{i64}$(SSS[2:0])

for(n = 0 ;n < 8; n++) {
  i = 64*n
  if (k1[n] != 0) {
    zmm1[i+63:i] = UpConvLoad$_{i64}$($m_t$+upSize*loadOffset)
    loadOffset++
    if ( ( ($m_t$ + upSize*loadOffset) % 64) == 0) {
      break
    }
  }
}
```

Flags Affected

None.

Memory Up-conversion: U$_{i64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

__m512i	_mm512_extloadunpacklo_epi64 (__m512i, void const*, _MM_UPCONV_EPI64_ENUM, int);
__m512i	_mm512_mask_extloadunpacklo_epi64 (__m512i, __mmask8, void const*, _MM_UPCONV_EPI64_ENUM, int);
__m512i	_mm512_loadunpacklo_epi64 (__m512i, void const*);
__m512i	_mm512_mask_loadunpacklo_epi64 (__m512i, __mmask8, void const*);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the UpConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the second operand is not a memory location.

VLOG2PS - Vector Logarithm Base-2 of Float32 Vector

Opcode	Instruction	Description
MVEX.512.66.0F38.W0 C9 /r	vlog2ps zmm1 {k1}, zmm2/m_t	Calculate logarithm from float32 vector zmm2/m_t and store the result in zmm1, under write-mask.

Description

Computes the element-by-element logarithm base-2 of the float32 vector on memory or float32 vector zmm2. The result is written into float32 vector zmm1.

1. 4ulp of relative error when the source value is within the intervals (0, 0.5) or (2, ∞]
2. absolute error less than 2^{-21} within the interval $[0.5, 2]$

For an input value of $+/-0$ the instruction returns $-\infty$ and sets the Divide-By-Zero flag (#Z). Negative numbers (including $-\infty$) should return the canonical NaN and set the Invalid flag (#I). Note however that this instruction treats input denormals as zeros of the same sign, so for denormal negative inputs it returns $-\infty$ and sets the Divide-By-Zero status flag. If any source element is NaN, the quietized NaN source value is returned for that element (and #I is raised for input sNaNs).

Current implementation of this instruction does not support any SwizzUpConv setting other than "no broadcast and no conversion"; any other SwizzUpConv setting will result in an Invalid Opcode exception.

log2_DX() function follows Table 6.25 when dealing with floating-point special numbers.

Input	Result	Comments
NaN	input qNaN	Raise #I flag if sNaN
$+\infty$	$+\infty$	
$+0$	$-\infty$	Raise #Z flag
-0	$-\infty$	Raise #Z flag
<0	NaN	Raise #I flag
$-\infty$	NaN	Raise #I flag
2^n	n	Exact integral result

Table 6.25: vlog2_DX() special floating-point values behavior

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
tmpSrc2[511:0] = zmm2/$m_t$

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags()    // SAE
}

for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] = vlog2_DX(tmpSrc2[i+31:i])
  }
}
```

SIMD Floating-Point Exceptions

Invalid, Zero.

Denormal Handling

Treat Input Denormals As Zeros :
YES

Flush Tiny Results To Zero :
YES

Register Swizzle

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	reserved	N/A
010	reserved	N/A
011	reserved	N/A
100	reserved	N/A
101	reserved	N/A
110	reserved	N/A
111	reserved	N/A

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_log2_ps (__m512);
__m512    _mm512_mask_log2_ps (__m512, __mmask16, __m512);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. This instruction does not support any SwizzUpConv different from the default value (no broadcast, no conversion). If SwizzUpConv function is set to any value different than "no action", then an Invalid Opcode fault is raised. This includes register swizzles.

VMOVAPD - Move Aligned Float64 Vector

Opcode	Instruction	Description
MVEX.512.66.0F.W1 28 /r	vmovapd zmm1 {k1}, $U_{f64}(m_t)$	Move float64 vector $U_{f64}(m_t)$ into vector zmm1, under write-mask.
MVEX.512.66.0F.W1 28 /r	vmovapd zmm1 {k1}, S_{f64}(zmm2)	Move float64 vector S_{f64}(zmm2) into vector zmm1, under write-mask.
MVEX.512.66.0F.W1 29 /r	vmovapd m_t {k1}, D_{f64}(zmm1)	Move float64 vector D_{f64}(zmm1) into m_t, under write-mask.

Description

Moves float64 vector result of the swizzle/broadcast/conversion process on memory or float64 vector zmm2 into float64 vector zmm1 or down-converts and stores float64 vector zmm2 into destination memory.

This instruction is write-masked, so only those elements with the corresponding bit(s) set in the vector mask (k1) register are computed and stored into register/memory. Elements in register/memory with the corresponding bit(s) clear in the vector mask register are maintained with the previous value.

Operation

DESTINATION IS A VECTOR OPERAND

```
if(source is a register operand) {
   if(MVEX.EH==1) {
      tmpSrc2[511:0] = zmm2[511:0]
   } else {
      tmpSrc2[511:0] = SwizzUpConvLoad$_{f64}$(zmm2)
   }
} else {
   tmpSrc2[511:0] = UpConvLoad$_{f64}$($m_t$)
}

for (n = 0; n < 8; n++) {
  if (k1[n] != 0) {
    i = 64*n
    zmm1[i+63:i] = tmpSrc2[i+63:i])
  }
}
```

DESTINATION IS A MEMORY OPERAND

```
downSize = DownConvStoreSizeOf$_{f64}$(SSS[2:0])

for(n = 0 ;n < 8; n++) {
```

```
        if (k1[n] != 0) {
          i = 64*n
          tmp = DownConvStore$_{f64}$(zmm1[i+63:i], SSS[2:0])
          if(downSize == 8) {
            MemStore($m_t$+8*n) = tmp[63:0]
          }
        }
      }
```

SIMD Floating-Point Exceptions

DESTINATION IS A VECTOR OPERAND: None.

DESTINATION IS A MEMORY OPERAND: None.

Memory Up-conversion: U$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Memory Down-conversion: D_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	64
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_mask_mov_pd (__m512d, __mmask8, __m512d);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VMOVAPS - Move Aligned Float32 Vector

Opcode	Instruction	Description
MVEX.512.0F.W0 28 /r	vmovaps zmm1 {k1}, $U_{f32}(m_t)$	Move float32 vector $U_{f32}(m_t)$ into vector zmm1, under write-mask.
MVEX.512.0F.W0 28 /r	vmovaps zmm1 {k1}, S_{f32}(zmm2)	Move float32 vector S_{f32}(zmm2) into vector zmm1, under write-mask.
MVEX.512.0F.W0 29 /r	vmovaps m_t {k1}, D_{f32}(zmm1)	Move float32 vector D_{f32}(zmm1) into m_t, under write-mask.

Description

Moves float32 vector result of the swizzle/broadcast/conversion process on memory or float32 vector zmm2 into float32 vector zmm1 or down-converts and stores float32 vector zmm2 into destination memory.

This instruction is write-masked, so only those elements with the corresponding bit(s) set in the vector mask (k1) register are computed and stored into register/memory. Elements in register/memory with the corresponding bit(s) clear in the vector mask register are maintained with the previous value.

Operation

DESTINATION IS A VECTOR OPERAND

```
if(source is a register operand) {
   if(MVEX.EH==1) {
      tmpSrc2[511:0] = zmm2[511:0]
   } else {
      tmpSrc2[511:0] = SwizzUpConvLoad$_{f32}$(zmm2)
   }
} else {
   tmpSrc2[511:0] = UpConvLoad$_{f32}$($m_t$)
}

for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] = tmpSrc2[i+31:i])
  }
}
```

DESTINATION IS A MEMORY OPERAND

```
downSize = DownConvStoreSizeOf$_{f32}$(SSS[2:0])

for(n = 0 ;n < 16; n++) {
```

```
        if (k1[n] != 0) {
          i = 32*n
          tmp = DownConvStore$_{f32}$(zmm1[i+31:i], SSS[2:0])
          if(downSize == 4) {
            MemStore($m_t$+4*n) = tmp[31:0]
          } else if(downSize == 2) {
            MemStore($m_t$+2*n) = tmp[15:0]
          } else if(downSize == 1) {
            MemStore($m_t$+n) = tmp[7:0]
          }
        }
      }
```

SIMD Floating-Point Exceptions

DESTINATION IS A VECTOR OPERAND: Invalid.

DESTINATION IS A MEMORY OPERAND: Overflow, Underflow, Invalid, Precision, Denormal.

Memory Up-conversion: U$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
101	sint8 to float32	[rax] {sint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Memory Down-conversion: D_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	64
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float32 to float16	zmm1 {float16}	32
100	float32 to uint8	zmm1 {uint8}	16
101	float32 to sint8	zmm1 {sint8}	16
110	float32 to uint16	zmm1 {uint16}	32
111	float32 to sint16	zmm1 {sint16}	32

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512   _mm512_mask_mov_ps (__m512, __mmask16, __m512);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VMOVDQA32 - Move Aligned Int32 Vector

Opcode	Instruction	Description
MVEX.512.66.0F.W0 6F /r	vmovdqa32 zmm1 {k1}, $U_{i32}(m_t)$	Move int32 vector $U_{i32}(m_t)$ into vector zmm1, under write-mask.
MVEX.512.66.0F.W0 6F /r	vmovdqa32 zmm1 {k1}, S_{i32}(zmm2)	Move int32 vector S_{i32}(zmm2) into vector zmm1, under write-mask.
MVEX.512.66.0F.W0 7F /r	vmovdqa32 m_t {k1}, D_{i32}(zmm1)	Move int32 vector D_{i32}(zmm1) into m_t, under write-mask.

Description

Moves int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm2 into int32 vector zmm1 or down-converts and stores int32 vector zmm2 into destination memory.

This instruction is write-masked, so only those elements with the corresponding bit(s) set in the vector mask (k1) register are computed and stored into register/memory. Elements in register/memory with the corresponding bit(s) clear in the vector mask register are maintained with the previous value.

Operation

DESTINATION IS A VECTOR OPERAND

```
if(source is a register operand) {
   if(MVEX.EH==1) {
      tmpSrc2[511:0] = zmm2[511:0]
   } else {
      tmpSrc2[511:0] = SwizzUpConvLoad$_{i32}$(zmm2)
   }
} else {
   tmpSrc2[511:0] = UpConvLoad$_{i32}$($m_t$)
}

for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] = tmpSrc2[i+31:i])
  }
}
```

DESTINATION IS A MEMORY OPERAND

```
downSize = DownConvStoreSizeOf$_{i32}$(SSS[2:0])
```

```
for(n = 0 ;n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    tmp = DownConvStore$_{i32}$(zmm1[i+31:i], SSS[2:0])
    if(downSize == 4) {
      MemStore($m_t$+4*n) = tmp[31:0]
    } else if(downSize == 2) {
      MemStore($m_t$+2*n) = tmp[15:0]
    } else if(downSize == 1) {
      MemStore($m_t$+n) = tmp[7:0]
    }
  }
}
```

Flags Affected

DESTINATION IS A VECTOR OPERAND: None.

DESTINATION IS A MEMORY OPERAND: None.

Memory Up-conversion: U$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Memory Down-conversion: D_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	64
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	uint32 to uint8	zmm1 {uint8}	16
101	sint32 to sint8	zmm1 {sint8}	16
110	uint32 to uint16	zmm1 {uint16}	32
111	sint32 to sint16	zmm1 {sint16}	32

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_mask_mov_epi32 (__m512i, __mmask16, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VMOVDQA64 - Move Aligned Int64 Vector

Opcode	Instruction	Description
MVEX.512.66.0F.W1 6F /r	vmovdqa64 zmm1 {k1}, $U_{i64}(m_t)$	Move int64 vector $U_{i64}(m_t)$ into vector zmm1, under write-mask.
MVEX.512.66.0F.W1 6F /r	vmovdqa64 zmm1 {k1}, S_{i64}(zmm2)	Move int64 vector S_{i64}(zmm2) into vector zmm1, under write-mask.
MVEX.512.66.0F.W1 7F /r	vmovdqa64 m_t {k1}, D_{i64}(zmm1)	Move int64 vector D_{i64}(zmm1) into m_t, under write-mask.

Description

Moves int64 vector result of the swizzle/broadcast/conversion process on memory or int64 vector zmm2 into int64 vector zmm1 or down-converts and stores int64 vector zmm2 into destination memory.

This instruction is write-masked, so only those elements with the corresponding bit(s) set in the vector mask (k1) register are computed and stored into register/memory. Elements in register/memory with the corresponding bit(s) clear in the vector mask register are maintained with the previous value.

Operation

DESTINATION IS A VECTOR OPERAND

```
if(source is a register operand) {
   if(MVEX.EH==1) {
      tmpSrc2[511:0] = zmm2[511:0]
   } else {
      tmpSrc2[511:0] = SwizzUpConvLoad$_{i64}$(zmm2)
   }
} else {
   tmpSrc2[511:0] = UpConvLoad$_{i64}$($m_t$)
}

for (n = 0; n < 8; n++) {
  if (k1[n] != 0) {
    i = 64*n
    zmm1[i+63:i] = tmpSrc2[i+63:i])
  }
}
```

DESTINATION IS A MEMORY OPERAND

```
downSize = DownConvStoreSizeOf$_{i64}$(SSS[2:0])
```

```
for(n = 0 ;n < 8; n++) {
  if (k1[n] != 0) {
    i = 64*n
    tmp = DownConvStore$_{i64}$(zmm1[i+63:i], SSS[2:0])
    if(downSize == 8) {
      MemStore($m_t$+8*n) = tmp[63:0]
    }
  }
}
```

Flags Affected

DESTINATION IS A VECTOR OPERAND: None.

DESTINATION IS A MEMORY OPERAND: None.

Memory Up-conversion: U$_{i64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{i64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Memory Down-conversion: D$_{i64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	64
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_mask_mov_epi64 (__m512i, __mmask8, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VMOVNRAPD - Store Aligned Float64 Vector With No-Read Hint

Opcode	Instruction	Description
MVEX.512.F3.0F.W1.EH0 29 /r	vmovnrapd m {k1}, D_{f64}(zmm1)	Store with No-Read hint float64 vector D_{f64}(zmm1) into m, under write-mask.

Description

Stores float64 vector zmm1 (or a down-converted version of it) into destination memory with a No-Read hint for the case the whole vector is going to be written into memory. This instruction is intended to speed up the case of stores in streaming kernels where we want to avoid wasting memory bandwidth by being forced to read the original content of entire cache lines from memory when we overwrite their whole contents completely.

In Knights Corner, this instruction is able to optimize memory bandwidth in case of a cache miss and avoid reading the original contents of the memory destination operand if the following conditions hold true:

- The instruction does not use a write-mask (MVEX.aaa=000).
- The instruction does not perform any kind of down-conversion (MVEX.SSS=000).

Note that this instruction is encoded by forcing MVEX.EH bit to 0. The Eviction Hint does not have any effect on this instruction.

The No-Read directive is intended as a performance hint and could be ignored by a given processor implementation.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are stored to memory. Elements in the destination memory vector with the corresponding bit clear in k1 register retain their previous value.

Operation

DESTINATION IS A MEMORY OPERAND

```
downSize = DownConvStoreSizeOf$_{f64}$(SSS[2:0])

for(n = 0 ;n < 8; n++) {
  if (k1[n] != 0) {
    i = 64*n
    tmp = DownConvStore$_{f64}$(zmm1[i+63:i], SSS[2:0])
    if(downSize == 8) {
      MemStore($m_t$+8*n) = tmp[63:0]
    }
  }
}
```

SIMD Floating-Point Exceptions

None.

Memory Down-conversion: D_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	64
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

```
void    _mm512_storenr_pd(void*, __m512d);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VMOVNRAPS - Store Aligned Float32 Vector With No-Read Hint

Opcode	Instruction	Description
MVEX.512.F2.0F.W0.EH0 29 /r	vmovnraps m {k1}, D_{f32}(zmm1)	Store with No-Read hint float32 vector D_{f32}(zmm1) into m, under write-mask.

Description

Stores float32 vector zmm1 (or a down-converted version of it) into destination memory with a No-Read hint for the case the whole vector is going to be written into memory. This instruction is intended to speed up the case of stores in streaming kernels where we want to avoid wasting memory bandwidth by being forced to read the original content of entire cache lines from memory when we overwrite their whole contents completely.

In Knights Corner, this instruction is able to optimize memory bandwidth in case of a cache miss and avoid reading the original contents of the memory destination operand if the following conditions hold true:

- The instruction does not use a write-mask (MVEX.aaa=000).
- The instruction does not perform any kind of down-conversion (MVEX.SSS=000).

Note that this instruction is encoded by forcing MVEX.EH bit to 0. The Eviction Hint does not have any effect on this instruction.

The No-Read directive is intended as a performance hint and could be ignored by a given processor implementation.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are stored to memory. Elements in the destination memory vector with the corresponding bit clear in k1 register retain their previous value.

Operation

DESTINATION IS A MEMORY OPERAND

```
downSize = DownConvStoreSizeOf$_{f32}$(SSS[2:0])

for(n = 0 ;n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    tmp = DownConvStore$_{f32}$(zmm1[i+31:i], SSS[2:0])
    if(downSize == 4) {
      MemStore($m_t$+4*n) = tmp[31:0]
    } else if(downSize == 2) {
      MemStore($m_t$+2*n) = tmp[15:0]
    } else if(downSize == 1) {
```

```
            MemStore($m_t$+n) = tmp[7:0]
        }
    }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Memory Down-conversion: D_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	64
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float32 to float16	zmm1 {float16}	32
100	float32 to uint8	zmm1 {uint8}	16
101	float32 to sint8	zmm1 {sint8}	16
110	float32 to uint16	zmm1 {uint16}	32
111	float32 to sint16	zmm1 {sint16}	32

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm512_storenr_ps(void*, __m512);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.

	If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.

VMOVNRNGOAPD - Non-globally Ordered Store Aligned Float64 Vector With No-Read Hint

Opcode	Instruction	Description
MVEX.512.F3.0F.W1.EH1 29 /r	vmovnrngoapd m {k1}, D_{f64}(zmm1)	Non-ordered Store with No-Read hint float64 vector D_{f64}(zmm1) into m, under write-mask.

Description

Stores float64 vector zmm1 (or a down-converted version of it) into destination memory with a No-Read hint for the case the whole vector is going to be written into memory, using a weakly-ordered memory consistency model (i.e. stores performed with these instruction are not globally ordered, and subsequent stores from the same thread can be observed before them).

This instruction is intended to speed up the case of stores in streaming kernels where we want to avoid wasting memory bandwidth by being forced to read the original content of entire cache lines from memory when we overwrite their whole contents completely. This instruction takes advantage of the weakly-ordered memory consistency model to increase the throughput at which this type of write operations can be performed. Due to the same reason, a fencing operation implemented with SFENCE, MFENCE or CPUID instructions should be used in conjunction with this instruction if multiple threads are reading/writing the memory operand location (note that Knights Corner does not implement SFENCE nor MFENCE).

In Knights Corner, this instruction is able to optimize memory bandwidth in case of a cache miss and avoid reading the original contents of the memory destination operand if the following conditions hold true:

- The instruction does not use a write-mask (MVEX.aaa=000).
- The instruction does not perform any kind of down-conversion (MVEX.SSS=000).

Note that this instruction is encoded by forcing MVEX.EH bit to 1. The Eviction Hint does not have any effect on this instruction.

The No-Read directive is intended as a performance hint and could be ignored by a given processor implementation.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are stored to memory. Elements in the destination memory vector with the corresponding bit clear in k1 register retain their previous value.

Operation

DESTINATION IS A MEMORY OPERAND

```
downSize = DownConvStoreSizeOf$_{f64}$(SSS[2:0])

for(n = 0 ;n < 8; n++) {
  if (k1[n] != 0) {
    i = 64*n
    tmp = DownConvStore$_{f64}$(zmm1[i+63:i], SSS[2:0])
    if(downSize == 8) {
      MemStore($m_t$+8*n) = tmp[63:0]
    }
  }
}
```

SIMD Floating-Point Exceptions

None.

Memory Down-conversion: D$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	64
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm512_storenrngo_pd(void*, __m512d);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VMOVNRNGOAPS - Non-globally Ordered Store Aligned Float32 Vector With No-Read Hint

Opcode	Instruction	Description
MVEX.512.F2.0F.W0.EH1 29 /r	vmovnrngoaps m {k1}, D_{f32}(zmm1)	Non-ordered Store with No-Read hint float32 vector D_{f32}(zmm1) into m, under write-mask.

Description

Stores float32 vector zmm1 (or a down-converted version of it) into destination memory with a No-Read hint for the case the whole vector is going to be written into memory, using a weakly-ordered memory consistency model (i.e. stores performed with these instruction are not globally ordered, and subsequent stores from the same thread can be observed before them).

This instruction is intended to speed up the case of stores in streaming kernels where we want to avoid wasting memory bandwidth by being forced to read the original content of entire cache lines from memory when we overwrite their whole contents completely. This instruction takes advantage of the weakly-ordered memory consistency model to increase the throughput at which this type of write operations can be performed. Due to the same reason, a fencing operation implemented with SFENCE, MFENCE or CPUID instructions should be used in conjunction with this instruction if multiple threads are reading/writing the memory operand location (note that Knights Corner does not implement SFENCE nor MFENCE).

In Knights Corner, this instruction is able to optimize memory bandwidth in case of a cache miss and avoid reading the original contents of the memory destination operand if the following conditions hold true:

- The instruction does not use a write-mask (MVEX.aaa=000).
- The instruction does not perform any kind of down-conversion (MVEX.SSS=000).

Note that this instruction is encoded by forcing MVEX.EH bit to 1. The Eviction Hint does not have any effect on this instruction.

The No-Read directive is intended as a performance hint and could be ignored by a given processor implementation.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are stored to memory. Elements in the destination memory vector with the corresponding bit clear in k1 register retain their previous value.

Operation

DESTINATION IS A MEMORY OPERAND

```
downSize = DownConvStoreSizeOf$_{f32}$(SSS[2:0])

for(n = 0 ;n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    tmp = DownConvStore$_{f32}$(zmm1[i+31:i], SSS[2:0])
    if(downSize == 4) {
      MemStore($m_t$+4*n) = tmp[31:0]
    } else if(downSize == 2) {
      MemStore($m_t$+2*n) = tmp[15:0]
    } else if(downSize == 1) {
      MemStore($m_t$+n) = tmp[7:0]
    }
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Memory Down-conversion: D$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	64
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float32 to float16	zmm1 {float16}	32
100	float32 to uint8	zmm1 {uint8}	16
101	float32 to sint8	zmm1 {sint8}	16
110	float32 to uint16	zmm1 {uint16}	32
111	float32 to sint16	zmm1 {sint16}	32

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm512_storenrngo_ps(void*, __m512);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VMULPD - Multiply Float64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F.W1 59 /r	vmulpd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Multiply float64 vector zmm2 and float64 vector S_{f64}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between float64 vector zmm2 and the float64 vector result of the swizzle/broadcast/conversion process on memory or float64 vector zmm3. The result is written into float64 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = zmm2[i+63:i] * tmpSrc3[i+63:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
(MXCSR.FZ)?  YES : NO
```

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_mul_pd (__m512d, __m512d);
__m512d    _mm512_mask_mul_pd (__m512d, __mmask8, __m512d, __m512d);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VMULPS - Multiply Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.0F.W0 59 /r	vmulps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Multiply float32 vector zmm2 and float32 vector S_{f32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between float32 vector zmm2 and the float32 vector result of the swizzle/broadcast/conversion process on memory or float32 vector zmm3. The result is written into float32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad_f32(zmm3/m_t)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = zmm2[i+31:i] * tmpSrc3[i+31:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
    (MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :

```
    (MXCSR.FZ)? YES : NO
```

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_mul_ps (__m512, __m512);
__m512    _mm512_mask_mul_ps (__m512, __mmask16, __m512, __m512);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPACKSTOREHD - Pack And Store Unaligned High From Int32 Vector

Opcode	Instruction	Description
MVEX.512.66.0F38.W0 D4 /r	vpackstorehd m_t {k1}, D_{i32}(zmm1)	Pack mask-enabled elements of int32 vector zmm1 to form an unaligned int32 stream, down-convert it and logically map the stream starting at $m_t - 64$, and store that portion of the stream that maps to the high 64-byte-aligned portion of the memory destination, under write-mask.

Description

Packs and down-converts the mask-enabled elements of int32 vector zmm1 into a byte/word/doubleword stream logically mapped starting at element-aligned address $(m_t - 64)$, and stores the high-64-byte elements of that stream (those elements of the stream that map at or after the first 64-byte-aligned address following $(m_t - 64)$, the high cache line in the current implementation). The length of the stream depends on the number of enabled masks, as elements disabled by the mask are not added to the stream.

The vpackstoreld instruction is used to store the part of the stream before the first 64-byte-aligned address preceding m_t.

The mask is not used as a write-mask for this instruction. Instead, the mask is used as an element selector, choosing which elements are added to the stream. The one similarity to a write-mask as used in the rest of this document is that the no-write-mask option (encoding 0) is available to select a mask of 0xFFFF for this instruction. For that reason, the notation and encoding are the same as for a write-mask.

In conjunction with vpackstoreld, this instruction is useful for packing data into a queue. Also in conjunction with vpackstoreld, it allows unaligned vector stores (that is, vector stores that are only element-wise , not vector-wise, aligned); just use a mask of 0xFFFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector store would be:

```
// assume memory location is pointed by register rax
vpackstoreld  [rax]    {k1}, v0
vpackstorehd  [rax+64] {k1}, v0
```

This instruction does not have subset support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element after down-conversion.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand. The instruction will not produce any #GP or #SS fault due to address canonicity nor #PF fault if the address is aligned to a 64-byte boundary. Additionally, A/D bits in the page table will not be updated.

Operation

```
storeOffset = 0
downSize = DownConvStoreSizeOf$_{i32}$(SSS[2:0])
foundNext64BytesBoundary = false

pointer = $m_t$ - 64
for (n = 0; n < 16; n++) {
   if(k1[n] != 0) {
      if (foundNext64BytesBoundary == false) {
         if ( ( (pointer + (storeOffset+1)*downSize) % 64) == 0 ) {
            foundNext64BytesBoundary = true
         }
      } else {
         i = 32*n
         tmp = DownConvStore$_{i32}$(zmm1[i+31:i], SSS[2:0])
         if(downSize == 4) {
           MemStore(pointer + storeOffset*4) = tmp[31:0]
         } else if(downSize == 2) {
           MemStore(pointer + storeOffset*2) = tmp[15:0]
         } else if(downSize == 1) {
           MemStore(pointer + storeOffset) = tmp[7:0]
         }
      }
      storeOffset++
   }
}
```

Flags Affected

None.

Memory Down-conversion: D$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	uint32 to uint8	zmm1 {uint8}	1
101	sint32 to sint8	zmm1 {sint8}	1
110	uint32 to uint16	zmm1 {uint16}	2
111	sint32 to sint16	zmm1 {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

```
void _mm512_extpackstorehi_epi32 (void*, __m512i, _MM_DOWNCONV_EPI32_ENUM, int);
void _mm512_mask_extpackstorehi_epi32 (void*, __mmask16, __m512i, _MM_DOWNCONV_EPI32_ENUM, int);
void _mm512_packstorehi_epi32 (void*, __m512i);
void _mm512_mask_packstorehi_epi32 (void*, __mmask16, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the DownConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the fist operand is not a memory location.

VPACKSTOREHPD - Pack And Store Unaligned High From Float64 Vector

Opcode	**Instruction**	**Description**
MVEX.512.66.0F38.W1 D5 /r	vpackstorehpd m_t {k1}, D_{f64}(zmm1)	Pack mask-enabled elements of float64 vector zmm1 to form an unaligned float64 stream, down-convert it and logically map the stream starting at $m_t - 64$, and store that portion of the stream that maps to the high 64-byte-aligned portion of the memory destination, under write-mask.

Description

Packs and down-converts the mask-enabled elements of float64 vector zmm1 into a float64 stream logically mapped starting at element-aligned address $(m_t - 64)$, and stores the high-64-byte elements of that stream (those elements of the stream that map at or after the first 64-byte-aligned address following $(m_t - 64)$, the high cache line in the current implementation). The length of the stream depends on the number of enabled masks, as elements disabled by the mask are not added to the stream.

The vpackstorelpd instruction is used to store the part of the stream before the first 64-byte-aligned address preceding m_t.

The mask is not used as a write-mask for this instruction. Instead, the mask is used as an element selector, choosing which elements are added to the stream. The one similarity to a write-mask as used in the rest of this document is that the no-write-mask option (encoding 0) is available to select a mask of 0xFF for this instruction. For that reason, the notation and encoding are the same as for a write-mask.

In conjunction with vpackstorelpd, this instruction is useful for packing data into a queue. Also in conjunction with vpackstorelpd, it allows unaligned vector stores (that is, vector stores that are only element-wise , not vector-wise, aligned); just use a mask of 0xFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector store would be:

```
// assume memory location is pointed by register rax
vpackstorelpd  [rax]    {k1}, v0
vpackstorehpd  [rax+64] {k1}, v0
```

This instruction does not have subset support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element after down-conversion.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand. The instruction will not produce any #GP or #SS fault due to address canonicity nor #PF fault if the address is aligned to a 64-byte boundary. Additionally, A/D bits in the page table will not be updated.

Operation

```
storeOffset = 0
downSize = DownConvStoreSizeOf$_{f64}$(SSS[2:0])
foundNext64BytesBoundary = false

pointer = $m_t$ - 64
for (n = 0; n < 8; n++) {
   if(k1[n] != 0) {
      if (foundNext64BytesBoundary == false) {
         if ( ( (pointer + (storeOffset+1)*downSize) % 64) == 0 ) {
            foundNext64BytesBoundary = true
         }
      } else {
         i = 64*n
         tmp = DownConvStore$_{f64}$(zmm1[i+63:i], SSS[2:0])
         if(downSize == 8) {
           MemStore(pointer + storeOffset*8) = tmp[63:0]
         }
      }
      storeOffset++
   }
}
```

SIMD Floating-Point Exceptions

None.

Memory Down-conversion: D$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm512_extpackstorehi_pd (void*, __m512d, _MM_DOWNCONV_PD_ENUM, int);
void _mm512_mask_extpackstorehi_pd (void*, __mmask8, __m512d, _MM_DOWNCONV_PD_ENUM, int);
void _mm512_packstorehi_pd (void*, __m512d);
void _mm512_mask_packstorehi_pd (void*, __mmask8, __m512d);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the DownConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the fist operand is not a memory location.

VPACKSTOREHPS - Pack And Store Unaligned High From Float32 Vector

Opcode	Instruction	Description
MVEX.512.66.0F38.W0 D5 /r	vpackstorehps m_t {k1}, D_{f32}(zmm1)	Pack mask-enabled elements of float32 vector zmm1 to form an unaligned float32 stream, down-convert it and logically map the stream starting at $m_t - 64$, and store that portion of the stream that maps to the high 64-byte-aligned portion of the memory destination, under write-mask.

Description

Packs and down-converts the mask-enabled elements of float32 vector zmm1 into a byte/word/doubleword stream logically mapped starting at element-aligned address $(m_t - 64)$, and stores the high-64-byte elements of that stream (those elements of the stream that map at or after the first 64-byte-aligned address following $(m_t - 64)$, the high cache line in the current implementation). The length of the stream depends on the number of enabled masks, as elements disabled by the mask are not added to the stream.

The vpackstorelps instruction is used to store the part of the stream before the first 64-byte-aligned address preceding m_t.

The mask is not used as a write-mask for this instruction. Instead, the mask is used as an element selector, choosing which elements are added to the stream. The one similarity to a write-mask as used in the rest of this document is that the no-write-mask option (encoding 0) is available to select a mask of 0xFFFF for this instruction. For that reason, the notation and encoding are the same as for a write-mask.

In conjunction with vpackstorelps, this instruction is useful for packing data into a queue. Also in conjunction with vpackstorelps, it allows unaligned vector stores (that is, vector stores that are only element-wise , not vector-wise, aligned); just use a mask of 0xFFFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector store would be:

```
// assume memory location is pointed by register rax
vpackstorelps  [rax]    {k1}, v0
vpackstorehps  [rax+64] {k1}, v0
```

This instruction does not have subset support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element after down-conversion.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand. The instruction will not produce any #GP or #SS fault due to address canonicity nor #PF fault if the address is aligned to a 64-byte boundary. Additionally, A/D bits in the page table will not be updated.

Operation

```
storeOffset = 0
downSize = DownConvStoreSizeOf$_{f32}$(SSS[2:0])
foundNext64BytesBoundary = false

pointer = $m_t$ - 64
for (n = 0; n < 16; n++) {
   if(k1[n] != 0) {
      if (foundNext64BytesBoundary == false) {
         if ( ( (pointer + (storeOffset+1)*downSize) % 64) == 0 ) {
            foundNext64BytesBoundary = true
         }
      } else {
         i = 32*n
         tmp = DownConvStore$_{f32}$(zmm1[i+31:i], SSS[2:0])
         if(downSize == 4) {
           MemStore(pointer + storeOffset*4) = tmp[31:0]
         } else if(downSize == 2) {
           MemStore(pointer + storeOffset*2) = tmp[15:0]
         } else if(downSize == 1) {
           MemStore(pointer + storeOffset) = tmp[7:0]
         }
      }
      storeOffset++
   }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Memory Down-conversion: D$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float32 to float16	zmm1 {float16}	2
100	float32 to uint8	zmm1 {uint8}	1
101	float32 to sint8	zmm1 {sint8}	1
110	float32 to uint16	zmm1 {uint16}	2
111	float32 to sint16	zmm1 {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

void	_mm512_extpackstorehi_ps (void*, __m512, _MM_DOWNCONV_PS_ENUM, int);
void	_mm512_mask_extpackstorehi_ps (void*, __mmask16, __m512, _MM_DOWNCONV_PS_ENUM, int);
void	_mm512_packstorehi_ps (void*, __m512);
void	_mm512_mask_packstorehi_ps (void*, __mmask16, __m512);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the DownConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the fist operand is not a memory location.

VPACKSTOREHQ - Pack And Store Unaligned High From Int64 Vector

Opcode	Instruction	Description
MVEX.512.66.0F38.W1 D4 /r	vpackstorehq m_t {k1}, D_{i64}(zmm1)	Pack mask-enabled elements of int64 vector zmm1 to form an unaligned int64 stream, down-convert it and logically map the stream starting at $m_t - 64$, and store that portion of the stream that maps to the high 64-byte-aligned portion of the memory destination, under write-mask.

Description

Packs and down-converts the mask-enabled elements of int64 vector zmm1 into a int64 stream logically mapped starting at element-aligned address $(m_t - 64)$, and stores the high-64-byte elements of that stream (those elements of the stream that map at or after the first 64-byte-aligned address following $(m_t - 64)$, the high cache line in the current implementation). The length of the stream depends on the number of enabled masks, as elements disabled by the mask are not added to the stream.

The vpackstorelq instruction is used to store the part of the stream before the first 64-byte-aligned address preceding m_t.

The mask is not used as a write-mask for this instruction. Instead, the mask is used as an element selector, choosing which elements are added to the stream. The one similarity to a write-mask as used in the rest of this document is that the no-write-mask option (encoding 0) is available to select a mask of 0xFF for this instruction. For that reason, the notation and encoding are the same as for a write-mask.

In conjunction with vpackstorelq, this instruction is useful for packing data into a queue. Also in conjunction with vpackstorelq, it allows unaligned vector stores (that is, vector stores that are only element-wise , not vector-wise, aligned); just use a mask of 0xFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector store would be:

```
// assume memory location is pointed by register rax
vpackstorelq  [rax]    {k1}, v0
vpackstorehq  [rax+64] {k1}, v0
```

This instruction does not have subset support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element after down-conversion.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand. The instruction will not produce any #GP or #SS fault due to address canonicity nor #PF fault if the address is aligned to a 64-byte boundary. Additionally, A/D bits in the page table will not be updated.

Operation

```
storeOffset = 0
downSize = DownConvStoreSizeOf$_{i64}$(SSS[2:0])
foundNext64BytesBoundary = false

pointer = $m_t$ - 64
for (n = 0; n < 8; n++) {
   if(k1[n] != 0) {
      if (foundNext64BytesBoundary == false) {
         if ( ( (pointer + (storeOffset+1)*downSize) % 64) == 0 ) {
            foundNext64BytesBoundary = true
         }
      } else {
         i = 64*n
         tmp = DownConvStore$_{i64}$(zmm1[i+63:i], SSS[2:0])
         if(downSize == 8) {
           MemStore(pointer + storeOffset*8) = tmp[63:0]
         }
      }
      storeOffset++
   }
}
```

Flags Affected

None.

Memory Down-conversion: D$_{i64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

```
void    _mm512_extpackstorehi_epi64 (void*, __m512i, _MM_DOWNCONV_EPI64_ENUM, int);
void    _mm512_mask_extpackstorehi_epi64 (void*, __mmask8, __m512i, _MM_DOWNCONV_EPI64_ENUM, int);
void    _mm512_packstorehi_epi64 (void*, __m512i);
void    _mm512_mask_packstorehi_epi64 (void*, __mmask8, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the DownConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the fist operand is not a memory location.

VPACKSTORELD - Pack and Store Unaligned Low From Int32 Vector

Opcode	**Instruction**	**Description**
MVEX.512.66.0F38.W0 D0 /r	vpackstoreld m_t {k1}, D_{i32}(zmm1)	Pack mask-enabled elements of int32 vector zmm1 to form an unaligned int32 stream, down-convert it and logically map the stream starting at m_t, and store that portion of the stream that maps to the low 64-byte-aligned portion of the memory destination, under write-mask.

Description

Packs and down-converts the mask-enabled elements of int32 vector zmm1 into a byte/word/doubleword stream logically mapped starting at element-aligned address m_t, and stores the low-64-byte elements of that stream (those elements of the stream that map before the first 64-byte-aligned address following m_t, the low cache line in the current implementation). The length of the stream depends on the number of enabled masks, as elements disabled by the mask are not added to the stream.

The vpackstorehd instruction is used to store the part of the stream at or after the first 64-byte-aligned address preceding m_t.

The mask is not used as a write-mask for this instruction. Instead, the mask is used as an element selector, choosing which elements are added to the stream. The one similarity to a write-mask as used in the rest of this document is that the no-write-mask option (encoding 0) is available to select a mask of 0xFFFF for this instruction. For that reason, the notation and encoding are the same as for a write-mask.

In conjunction with vpackstorehd, this instruction is useful for packing data into into a queue. Also in conjunction with vpackstorehd, it allows unaligned vector stores (that is, vector stores that are only element-wise, not vector-wise, aligned); just use a mask of 0xFFFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector store would be:

```
// assume memory location is pointed by register rax
vpackstoreld  [rax]    {k1}, v0
vpackstorehd  [rax+64] {k1}, v0
```

This instruction does not have subset support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element after down-conversion.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand.

Operation

```
storeOffset = 0
downSize = DownConvStoreSizeOf$_{i32}$(SSS[2:0])

for(n = 0 ;n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    tmp = DownConvStore$_{i32}$(zmm1[i+31:i], SSS[2:0])
    if(downSize == 4) {
      MemStore($m_t$+4*storeOffset) = tmp[31:0]
    } else if(downSize == 2) {
      MemStore($m_t$+2*storeOffset) = tmp[15:0]
    } else if(downSize == 1) {
      MemStore($m_t$+storeOffset) = tmp[7:0]
    }
    storeOffset++
    if ((($m_t$ + downSize*storeOffset) % 64) == 0) {
      break
    }
  }
}
```

Flags Affected

None.

Memory Down-conversion: D$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	uint32 to uint8	zmm1 {uint8}	1
101	sint32 to sint8	zmm1 {sint8}	1
110	uint32 to uint16	zmm1 {uint16}	2
111	sint32 to sint16	zmm1 {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

void	_mm512_extpackstorelo_epi32 (void*, __m512i, _MM_DOWNCONV_EPI32_ENUM, int);
void	_mm512_mask_extpackstorelo_epi32 (void*, __mmask16, __m512i, _MM_DOWNCONV_EPI32_ENUM, int);
void	_mm512_packstorelo_epi32 (void*, __m512i);
void	_mm512_mask_packstorelo_epi32 (void*, __mmask16, __m512i);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the DownConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the fist operand is not a memory location.

VPACKSTORELPD - Pack and Store Unaligned Low From Float64 Vector

Opcode	**Instruction**	**Description**
MVEX.512.66.0F38.W1 D1 /r	vpackstorelpd m_t {k1}, D_{f64}(zmm1)	Pack mask-enabled elements of float64 vector zmm1 to form an unaligned float64 stream, down-convert it and logically map the stream starting at m_t, and store that portion of the stream that maps to the low 64-byte-aligned portion of the memory destination, under write-mask.

Description

Packs and down-converts the mask-enabled elements of float64 vector zmm1 into a float64 stream logically mapped starting at element-aligned address m_t, and stores the low-64-byte elements of that stream (those elements of the stream that map before the first 64-byte-aligned address following m_t, the low cache line in the current implementation). The length of the stream depends on the number of enabled masks, as elements disabled by the mask are not added to the stream.

The vpackstorehpd instruction is used to store the part of the stream at or after the first 64-byte-aligned address preceding m_t.

The mask is not used as a write-mask for this instruction. Instead, the mask is used as an element selector, choosing which elements are added to the stream. The one similarity to a write-mask as used in the rest of this document is that the no-write-mask option (encoding 0) is available to select a mask of 0xFF for this instruction. For that reason, the notation and encoding are the same as for a write-mask.

In conjunction with vpackstorehpd, this instruction is useful for packing data into into a queue. Also in conjunction with vpackstorehpd, it allows unaligned vector stores (that is, vector stores that are only element-wise, not vector-wise, aligned); just use a mask of 0xFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector store would be:

```
// assume memory location is pointed by register rax
vpackstorelpd  [rax]    {k1}, v0
vpackstorehpd  [rax+64] {k1}, v0
```

This instruction does not have subset support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element after down-conversion.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand.

Operation

```
storeOffset = 0
downSize = DownConvStoreSizeOf_{f64}(SSS[2:0])

for(n = 0 ;n < 8; n++) {
  if (k1[n] != 0) {
    i = 64*n
    tmp = DownConvStore_{f64}(zmm1[i+63:i], SSS[2:0])
    if(downSize == 8) {
      MemStore($m_t$+8*storeOffset) = tmp[63:0]
    }
    storeOffset++
    if ((($m_t$ + downSize*storeOffset) % 64) == 0) {
      break
    }
  }
}
```

SIMD Floating-Point Exceptions

None.

Memory Down-conversion: D_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm512_extpackstorelo_pd (void*, __m512d, _MM_DOWNCONV_PD_ENUM, int);
void _mm512_mask_extpackstorelo_pd (void*, __mmask8, __m512d, _MM_DOWNCONV_PD_ENUM, int);
void _mm512_packstorelo_pd (void*, __m512d);
void _mm512_mask_packstorelo_pd (void*, __mmask8, __m512d);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the DownConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the fist operand is not a memory location.

VPACKSTORELPS - Pack and Store Unaligned Low From Float32 Vector

Opcode	Instruction	Description
MVEX.512.66.0F38.W0 D1 /r	vpackstorelps m_t {k1}, D_{f32}(zmm1)	Pack mask-enabled elements of float32 vector zmm1 to form an unaligned float32 stream, down-convert it and logically map the stream starting at m_t, and store that portion of the stream that maps to the low 64-byte-aligned portion of the memory destination, under write-mask.

Description

Packs and down-converts the mask-enabled elements of float32 vector zmm1 into a byte/word/doubleword stream logically mapped starting at element-aligned address m_t, and stores the low-64-byte elements of that stream (those elements of the stream that map before the first 64-byte-aligned address following m_t, the low cache line in the current implementation). The length of the stream depends on the number of enabled masks, as elements disabled by the mask are not added to the stream.

The vpackstorehps instruction is used to store the part of the stream at or after the first 64-byte-aligned address preceding m_t.

The mask is not used as a write-mask for this instruction. Instead, the mask is used as an element selector, choosing which elements are added to the stream. The one similarity to a write-mask as used in the rest of this document is that the no-write-mask option (encoding 0) is available to select a mask of 0xFFFF for this instruction. For that reason, the notation and encoding are the same as for a write-mask.

In conjunction with vpackstorehps, this instruction is useful for packing data into into a queue. Also in conjunction with vpackstorehps, it allows unaligned vector stores (that is, vector stores that are only element-wise, not vector-wise, aligned); just use a mask of 0xFFFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector store would be:

```
// assume memory location is pointed by register rax
vpackstorelps  [rax]    {k1}, v0
vpackstorehps  [rax+64] {k1}, v0
```

This instruction does not have subset support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element after down-conversion.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand.

Operation

```
storeOffset = 0
downSize = DownConvStoreSizeOf$_{f32}$(SSS[2:0])

for(n = 0 ;n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    tmp = DownConvStore$_{f32}$(zmm1[i+31:i], SSS[2:0])
    if(downSize == 4) {
      MemStore($m_t$+4*storeOffset) = tmp[31:0]
    } else if(downSize == 2) {
      MemStore($m_t$+2*storeOffset) = tmp[15:0]
    } else if(downSize == 1) {
      MemStore($m_t$+storeOffset) = tmp[7:0]
    }
    storeOffset++
    if ((($m_t$ + downSize*storeOffset) % 64) == 0) {
      break
    }
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Memory Down-conversion: D$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float32 to float16	zmm1 {float16}	2
100	float32 to uint8	zmm1 {uint8}	1
101	float32 to sint8	zmm1 {sint8}	1
110	float32 to uint16	zmm1 {uint16}	2
111	float32 to sint16	zmm1 {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

void	_mm512_extpackstorelo_ps (void*, __m512, _MM_DOWNCONV_PS_ENUM, int);
void	_mm512_mask_extpackstorelo_ps (void*, __mmask16, __m512, _MM_DOWNCONV_PS_ENUM, int);
void	_mm512_packstorelo_ps (void*, __m512);
void	_mm512_mask_packstorelo_ps (void*, __mmask16, __m512);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the DownConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the fist operand is not a memory location.

VPACKSTORELQ - Pack and Store Unaligned Low From Int64 Vector

Opcode	Instruction	Description
MVEX.512.66.0F38.W1 D0 /r	vpackstorelq m_t {k1}, D_{i64}(zmm1)	Pack mask-enabled elements of int64 vector zmm1 to form an unaligned int64 stream, down-convert it and logically map the stream starting at m_t, and store that portion of the stream that maps to the low 64-byte-aligned portion of the memory destination, under write-mask.

Description

Packs and down-converts the mask-enabled elements of int64 vector zmm1 into a int64 stream logically mapped starting at element-aligned address m_t, and stores the low-64-byte elements of that stream (those elements of the stream that map before the first 64-byte-aligned address following m_t, the low cache line in the current implementation). The length of the stream depends on the number of enabled masks, as elements disabled by the mask are not added to the stream.

The vpackstorehq instruction is used to store the part of the stream at or after the first 64-byte-aligned address preceding m_t.

The mask is not used as a write-mask for this instruction. Instead, the mask is used as an element selector, choosing which elements are added to the stream. The one similarity to a write-mask as used in the rest of this document is that the no-write-mask option (encoding 0) is available to select a mask of 0xFF for this instruction. For that reason, the notation and encoding are the same as for a write-mask.

In conjunction with vpackstorehq, this instruction is useful for packing data into into a queue. Also in conjunction with vpackstorehq, it allows unaligned vector stores (that is, vector stores that are only element-wise, not vector-wise, aligned); just use a mask of 0xFF or no write-mask for this purpose. The typical instruction sequence to perform an unaligned vector store would be:

```
// assume memory location is pointed by register rax
vpackstorelq  [rax]    {k1}, v0
vpackstorehq  [rax+64] {k1}, v0
```

This instruction does not have subset support.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element after down-conversion.

Note that the address reported by a page fault is the beggining of the 64-byte cache line boundary containing the memory operand.

Operation

```
storeOffset = 0
downSize = DownConvStoreSizeOf$_{i64}$(SSS[2:0])

for(n = 0 ;n < 8; n++) {
  if (k1[n] != 0) {
    i = 64*n
    tmp = DownConvStore$_{i64}$(zmm1[i+63:i], SSS[2:0])
    if(downSize == 8) {
      MemStore($m_t$+8*storeOffset) = tmp[63:0]
    }
    storeOffset++
    if ((($m_t$ + downSize*storeOffset) % 64) == 0) {
      break
    }
  }
}
```

Flags Affected

None.

Memory Down-conversion: D$_{i64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm512_extpackstorelo_epi64 (void*, __m512i, _MM_DOWNCONV_EPI64_ENUM, int);
void _mm512_mask_extpackstorelo_epi64 (void*, __mmask8, __m512i, _MM_DOWNCONV_EPI64_ENUM, int);
void _mm512_packstorelo_epi64 (void*, __m512i);
void _mm512_mask_packstorelo_epi64 (void*, __mmask8, __m512i);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to element-wise data granularity dictated by the DownConv.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If the fist operand is not a memory location.

VPADCD - Add Int32 Vectors with Carry

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W0 5C /r	vpadcd zmm1 {k1}, k2, S_{i32}(zmm3/m_t)	Add int32 vector S_{i32}(zmm3/m_t), vector mask register k2 and int32 vector zmm1 and store the result in zmm1, and the carry of the sum in k2, under write-mask.

Description

Performs an element-by-element three-input addition between int32 vector zmm1, the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3, and the corresponding bit of k2. The result is written into int32 vector zmm1.

In addition, the carry from the sum for the n-th element is written into the n-th bit of vector mask k2.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1 and k2. Elements in zmm1 and k2 with the corresponding bit clear in k1 retain their previous value.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // integer operation
    tmpCarry = Carry(zmm1[i+31:i] + k2[n] + tmpSrc3[i+31:i])
    zmm1[i+31:i] = zmm1[i+31:i] + k2[n] + tmpSrc3[i+31:i]
    k2[n] = tmpCarry
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i   _mm512_adc_epi32(__m512i, __mmask16, __m512i, __mmask16*);
__m512i   _mm512_mask_adc_epi32(__m512i, __mmask16, __mmask16, __m512i,
          __mmask16*);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPADDD - Add Int32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F.W0 FE /r	vpaddd zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Add int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element addition between int32 vector zmm2 and the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3. The result is written into int32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // integer operation
    zmm1[i+31:i] = zmm2[i+31:i] + tmpSrc3[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_add_epi32 (__m512i, __m512i);
__m512i    _mm512_mask_add_epi32 (__m512i, __mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPADDSETCD - Add Int32 Vectors and Set Mask to Carry

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W0 5D /r	vpaddsetcd zmm1 {k1}, k2, S_{i32}(zmm3/m_t)	Add int32 vector zmm1 and int32 vector S_{i32}(zmm3/m_t) and store the sum in zmm1 and the carry from the sum in k2, under write-mask.

Description

Performs an element-by-element addition between int32 vector zmm1 and the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3. The result is written into int32 vector zmm1.

In addition, the carry from the sum for the n-th element is written into the n-th bit of vector mask k2.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1 and k2. Elements in zmm1 and k2 with the corresponding bit clear in k1 retain their previous value.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // integer operation
    k2[n] = Carry(zmm1[i+31:i] + tmpSrc3[i+31:i])
    zmm1[i+31:i] = zmm1[i+31:i] + tmpSrc3[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_addsetc_epi32 (__m512i, __m512i, __mmask16*);
__m512i    _mm512_mask_addsetc_epi32 (__m512i, __mmask16,__mmask16, __m512i,
           __mmask16*);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPADDSETSD - Add Int32 Vectors and Set Mask to Sign

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W0 CD /r	vpaddsetsd zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Add int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the sum in zmm1 and the sign from the sum in k1, under write-mask.

Description

Performs an element-by-element addition between int32 vector zmm2 and the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3. The result is written into int32 vector zmm1.

In addition, the sign of the result for the n-th element is written into the n-th bit of vector mask k1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    zmm1[i+31:i] = zmm2[i+31:i] + tmpSrc3[i+31:i]
    k1[n] = zmm1[i+31]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i   _mm512_addsets_epi32 (__m512i, __m512i, __mmask16*);
__m512i   _mm512_mask_addsets_epi32 (__m512i, __mmask16, __m512i, __m512i,
          __mmask16*);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If any memory operand linear address is not aligned to 4-byte data granularity.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If no write mask is provided or selected write-mask is k0.

VPANDD - Bitwise AND Int32 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F.W0 DB /r	vpandd zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Perform a bitwise AND between int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element bitwise AND between int32 vector zmm2 and the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3. The result is written into int32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] = zmm2[i+31:i] & tmpSrc3[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_and_epi32(__m512i, __m512i);
__m512i    _mm512_mask_and_epi32(__m512i, __mmask16,__m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPANDND - Bitwise AND NOT Int32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F.W0 DF /r	vpandnd zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Perform a bitwise AND between NOT int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element bitwise AND between NOT int32 vector zmm2 and the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3. The result is written into int32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] = (~(zmm2[i+31:i])) & tmpSrc3[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_andnot_epi32 (__m512i, __m512i);
__m512i    _mm512_mask_andnot_epi32 (__m512i, __mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPANDNQ - Bitwise AND NOT Int64 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F.W1 DF /r	vpandnq zmm1 {k1}, zmm2, S_{i64}(zmm3/m_t)	Perform a bitwise AND between NOT int64 vector zmm2 and int64 vector S_{i64}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element bitwise AND between NOT int64 vector zmm2 and the int64 vector result of the swizzle/broadcast/conversion process on memory or int64 vector zmm3. The result is written into int64 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    zmm1[i+63:i] = (~(zmm2[i+63:i])) & tmpSrc3[i+63:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S_{i64}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_andnot_epi64(__m512i, __m512i);
__m512i    _mm512_mask_andnot_epi64(__m512i, __mmask8, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPANDQ - Bitwise AND Int64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F.W1 DB /r	vpandq zmm1 {k1}, zmm2, S_{i64}(zmm3/m_t)	Perform a bitwise AND between int64 vector zmm2 and int64 vector S_{i64}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element bitwise AND between int64 vector zmm2 and the int64 vector result of the swizzle/broadcast/conversion process on memory or int64 vector zmm3. The result is written into int32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    zmm1[i+63:i] = zmm2[i+63:i] & tmpSrc3[i+63:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S_{i64}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i   _mm512_and_epi64(__m512i, __m512i);
__m512i   _mm512_mask_and_epi64(__m512i, __mmask8, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPBLENDMD - Blend Int32 Vectors using the Instruction Mask

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 64 /r	vpblendmd zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Blend int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element blending between int32 vector zmm2 and the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3, using the instruction mask as selector. The result is written into int32 vector zmm1.

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

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = tmpSrc3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(tmpSrc3/$m_t$)
}

for (n = 0; n < 16; n++) {
   if(k1[n]==1 or *no write-mask*) {
      zmm1[i+31:i] = tmpSrc3[i+31:i]
   } else {
      zmm1[i+31:i] = zmm2[i+31:i]
   }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_mask_blend_epi32 (__mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPBLENDMQ - Blend Int64 Vectors using the Instruction Mask

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W1 64 /r	vpblendmq zmm1 {k1}, zmm2, S_{i64}(zmm3/m_t)	Blend int64 vector zmm2 and int64 vector S_{i64}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element blending between int64 vector zmm2 and the int64 vector result of the swizzle/broadcast/conversion process on memory or int64 vector zmm3, using the instruction mask as selector. The result is written into int64 vector zmm1.

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

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = tmpSrc3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i64}$(tmpSrc3/$m_t$)
}

for (n = 0; n < 8; n++) {
   if(k1[n]==1 or *no write-mask*) {
      zmm1[i+63:i] = tmpSrc3[i+63:i]
   } else {
      zmm1[i+63:i] = zmm2[i+63:i]
   }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S_{i64}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_mask_blend_epi64 (__mmask8, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPBROADCASTD - Broadcast Int32 Vector

Opcode	Instruction	Description
MVEX.512.66.0F38.W0 58 /r	vpbroadcastd zmm1 {k1}, $U_{i32}(m_t)$	Broadcast int32 vector $U_{i32}(m_t)$ into vector zmm1, under write-mask.

Description

The 1, 2, or 4 bytes (depending on the conversion and broadcast in effect) at memory address m_t are broadcast and/or converted to a int32 vector. The result is written into int32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
// {1to16}
tmpSrc2[31:0] = UpConvLoad$_{i32}$($m_t$)
for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] = tmpSrc2[31:0]
  }
}
```

Flags Affected

None.

Memory Up-conversion: U_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	1
101	sint8 to sint32	[rax] {sint8}	1
110	uint16 to uint32	[rax] {uint16}	2
111	sint16 to sint32	[rax] {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

__m512i	_mm512_extload_epi32(void const*,_MM_UPCONV_EPI32_ENUM, _MM_BROADCAST32_ENUM, int);
__m512i	_mm512_mask_extload_epi32(__m512i, __mmask16, void const*,_MM_UPCONV_EPI32_ENUM, _MM_BROADCAST32_ENUM, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPBROADCASTQ - Broadcast Int64 Vector

Opcode	**Instruction**	**Description**
MVEX.512.66.0F38.W1 59 /r	vpbroadcastq zmm1 {k1}, $U_{i64}(m_t)$	Broadcast int64 vector $U_{i64}(m_t)$ into vector zmm1, under write-mask.

Description

The 8 bytes at memory address m_t are broadcast to a int64 vector. The result is written into int64 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
// {1to8}
tmpSrc2[63:0] = UpConvLoad$_{i64}$($m_t$)
for (n = 0; n < 8; n++) {
  if (k1[n] != 0) {
    i = 64*n
    zmm1[i+63:i] = tmpSrc2[63:0]
  }
}
```

Flags Affected

None.

Memory Up-conversion: U_{i64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

__m512i	_mm512_extload_epi64(void const*,_MM_UPCONV_EPI64_ENUM, _MM_BROADCAST64_ENUM, int);
__m512i	_mm512_mask_extload_epi64(__m512i, __mmask16, void const*,_MM_UPCONV_EPI64_ENUM, _MM_BROADCAST64_ENUM, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPCMPD - Compare Int32 Vectors and Set Vector Mask

Opcode	Instruction	Description
MVEX.NDS.512.66.0F3A.W0 1F /r ib	vpcmpd k2 {k1}, zmm1, S_{i32}(zmm2/m_t), imm8	Compare between int32 vector zmm1 and int32 vector S_{i32}(zmm2/m_t) and store the result in k2, under write-mask.

Description

Performs an element-by-element comparison between int32 vector zmm1 and the int32 vector result of the swizzle/broadcast/conversion from memory or int32 vector zmm2. The result is written into vector mask k2.

The write-mask does not perform the normal write-masking function for this instruction. While it does enable/disable comparisons, it does not block updating of the destination; instead, if a write-mask bit is 0, the corresponding destination bit is set to 0. Nonetheless, the operation is similar enough so that it makes sense to use the usual write-mask notation. This mode of operation is desirable because the result will be used directly as a write-mask, rather than the normal case where the result is used with a separate write-mask that keeps the masked elements inactive.

Immediate Format

	Comparison Type	I_2	I_1	I_0
eq	Equal	0	0	0
lt	Less than	0	0	1
le	Less than or Equal	0	1	0
neq	Not Equal	1	0	0
nlt	Not Less than	1	0	1
nle	Not Less than or Equal	1	1	0

Operation

```
switch (IMM8[2:0]) {
    case 0: OP ← EQ; break;
    case 1: OP ← LT; break;
    case 2: OP ← LE; break;
    case 4: OP ← NEQ; break;
    case 5: OP ← NLT; break;
```

```
    case 6: OP ← NLE; break;
    default: Reserved; break;
}

if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{i32}$(zmm2/$m_t$)
}

for (n = 0; n < 16; n++) {
  k2[n] = 0
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    k2[n] = (zmm1[i+31:i] OP tmpSrc2[i+31:i]) ? 1 : 0
  }
}
```

Instruction Pseudo-ops

Compilers and assemblers may implement the following pseudo-ops in addition to the standard instruction op:

Pseudo-Op	Implementation
vpcmpeqd k2 {k1}, zmm1, S_i(zmm2/m_t)	vcmpd k2 {k1}, zmm1, S_i(zmm2/m_t), {eq}
vpcmpltd k2 {k1}, zmm1, S_i(zmm2/m_t)	vcmpd k2 {k1}, zmm1, S_i(zmm2/m_t), {lt}
vpcmpled k2 {k1}, zmm1, S_i(zmm2/m_t)	vcmpd k2 {k1}, zmm1, S_i(zmm2/m_t), {le}
vpcmpneqd k2 {k1}, zmm1, S_i(zmm2/m_t)	vcmpd k2 {k1}, zmm1, S_i(zmm2/m_t), {neq}
vpcmpnltd k2 {k1}, zmm1, S_i(zmm2/m_t)	vcmpd k2 {k1}, zmm1, S_i(zmm2/m_t), {nlt}
vpcmpnled k2 {k1}, zmm1, S_i(zmm2/m_t)	vcmpd k2 {k1}, zmm1, S_i(zmm2/m_t), {nle}

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

__mmask16	_mm512_cmp_epi32_mask(__m512i, __m512i, const _MM_CMPINT_ENUM);
__mmask16	_mm512_mask_cmp_epi32_mask(__mmask16, __m512i, __m512i, const _MM_CMPINT_ENUM);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPCMPEQD - Compare Equal Int32 Vectors and Set Vector Mask

Opcode	Instruction	Description
MVEX.NDS.512.66.0F.W0 76 /r	vpcmpeqd k2 {k1}, zmm1, S_{i32}(zmm2/m_t)	Compare Equal between int32 vector zmm1 and int32 vector S_{i32}(zmm2/m_t), and set vector mask k2 to reflect the zero/non-zero status of each element of the result, under write-mask.

Description

Performs an element-by-element compare for equality between int32 vector zmm1 and the int32 vector result of the swizzle/broadcast/conversion from memory or int32 vector zmm2. The result is written into vector mask k2.

The write-mask does not perform the normal write-masking function for this instruction. While it does enable/disable comparisons, it does not block updating of the destination; instead, if a write-mask bit is 0, the corresponding destination bit is set to 0. Nonetheless, the operation is similar enough so that it makes sense to use the usual write-mask notation. This mode of operation is desirable because the result will be used directly as a write-mask, rather than the normal case where the result is used with a separate write-mask that keeps the masked elements inactive.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{i32}$(zmm2/$m_t$)
}

for (n = 0; n < 16; n++) {
  k2[n] = 0
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    k2[n] = (zmm1[i+31:i] == tmpSrc2[i+31:i]) ? 1 : 0
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__mmask16    _mm512_cmpeq_epi32_mask (__m512i, __m512i);
__mmask16    _mm512_mask_cmpeq_epi32_mask (__mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPCMPGTD - Compare Greater Than Int32 Vectors and Set Vector Mask

Opcode	Instruction	Description
MVEX.NDS.512.66.0F.W0 66 /r	vpcmpgtd k2 {k1}, zmm1, S_{i32}(zmm2/m_t)	Compare Greater between int32 vector zmm1 and int32 vector S_{i32}(zmm2/m_t), and set vector mask k2 to reflect the zero/non-zero status of each element of the result, under write-mask.

Description

Performs an element-by-element compare for the greater value of int32 vector zmm1 and the int32 vector result of the swizzle/broadcast/conversion from memory or int32 vector zmm2. The result is written into vector mask k2.

The write-mask does not perform the normal write-masking function for this instruction. While it does enable/disable comparisons, it does not block updating of the destination; instead, if a write-mask bit is 0, the corresponding destination bit is set to 0. Nonetheless, the operation is similar enough so that it makes sense to use the usual write-mask notation. This mode of operation is desirable because the result will be used directly as a write-mask, rather than the normal case where the result is used with a separate write-mask that keeps the masked elements inactive.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{i32}$(zmm2/$m_t$)
}

for (n = 0; n < 16; n++) {
  k2[n] = 0
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    k2[n] = (zmm1[i+31:i] > tmpSrc2[i+31:i]) ? 1 : 0
  }
}
```

Flags Affected

None.

Memory Up-conversion: S$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__mmask16    _mm512_cmpgt_epi32_mask (__m512i, __m512i);
__mmask16    _mm512_mask_cmpgt_epi32_mask (__mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPCMPLTD - Compare Less Than Int32 Vectors and Set Vector Mask

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 74 /r	vpcmpltd k2 {k1}, zmm1, S_{i32}(zmm2/m_t)	Compare Less between int32 vector zmm1 and int32 vector S_{i32}(zmm2/m_t), and set vector mask k2 to reflect the zero/non-zero status of each element of the result, under write-mask.

Description

Performs an element-by-element compare for the lesser value of int32 vector zmm1 and the int32 vector result of the swizzle/broadcast/conversion from memory or int32 vector zmm2. The result is written into vector mask k2.

The write-mask does not perform the normal write-masking function for this instruction. While it does enable/disable comparisons, it does not block updating of the destination; instead, if a write-mask bit is 0, the corresponding destination bit is set to 0. Nonetheless, the operation is similar enough so that it makes sense to use the usual write-mask notation. This mode of operation is desirable because the result will be used directly as a write-mask, rather than the normal case where the result is used with a separate write-mask that keeps the masked elements inactive.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{i32}$(zmm2/$m_t$)
}

for (n = 0; n < 16; n++) {
  k2[n] = 0
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    k2[n] = (zmm1[i+31:i] < tmpSrc2[i+31:i]) ? 1 : 0
  }
}
```

Flags Affected

None.

Memory Up-conversion: S$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__mmask16    _mm512_cmplt_epi32_mask (__m512i, __m512i);
__mmask16    _mm512_mask_cmplt_epi32_mask (__mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPCMPUD - Compare Uint32 Vectors and Set Vector Mask

Opcode	Instruction	Description
MVEX.NDS.512.66.0F3A.W0 1E /r ib	vpcmpud k2 {k1}, zmm1, S_{i32}(zmm2/m_t), imm8	Compare between uint32 vector zmm1 and uint32 vector S_{i32}(zmm2/m_t) and store the result in k2, under write-mask.

Description

Performs an element-by-element comparison between uint32 vector zmm1 and the uint32 vector result of the swizzle/broadcast/conversion from memory or uint32 vector zmm2. The result is written into vector mask k2.

The write-mask does not perform the normal write-masking function for this instruction. While it does enable/disable comparisons, it does not block updating of the destination; instead, if a write-mask bit is 0, the corresponding destination bit is set to 0. Nonetheless, the operation is similar enough so that it makes sense to use the usual write-mask notation. This mode of operation is desirable because the result will be used directly as a write-mask, rather than the normal case where the result is used with a separate write-mask that keeps the masked elements inactive.

Immediate Format

	Comparison Type	I_2	I_1	I_0
eq	Equal	0	0	0
lt	Less than	0	0	1
le	Less than or Equal	0	1	0
neq	Not Equal	1	0	0
nlt	Not Less than	1	0	1
nle	Not Less than or Equal	1	1	0

Operation

```
switch (IMM8[2:0]) {
    case 0: OP ← EQ; break;
    case 1: OP ← LT; break;
    case 2: OP ← LE; break;
    case 4: OP ← NEQ; break;
```

```
        case 5: OP ← NLT; break;
        case 6: OP ← NLE; break;
        default: Reserved; break;
}

if(source is a register operand and MVEX.EH bit is 1) {
    tmpSrc2[511:0] = zmm2[511:0]
} else {
    tmpSrc2[511:0] = SwizzUpConvLoad$_{i32}$(zmm2/$m_t$)
}

for (n = 0; n < 16; n++) {
  k2[n] = 0
  if(k1[n] != 0) {
    i = 32*n
    // unsigned integer operation
    k2[n] = (zmm1[i+31:i] OP tmpSrc2[i+31:i]) ? 1 : 0
  }
}
```

Instruction Pseudo-ops

Compilers and assemblers may implement the following pseudo-ops in addition to the standard instruction op:

Pseudo-Op	Implementation
vpcmpequd k2 {k1}, zmm1, S_i(zmm2/m_t)	vcmpud k2 {k1}, zmm1, S_i(zmm2/m_t), {eq}
vpcmpltud k2 {k1}, zmm1, S_i(zmm2/m_t)	vcmpud k2 {k1}, zmm1, S_i(zmm2/m_t), {lt}
vpcmpleud k2 {k1}, zmm1, S_i(zmm2/m_t)	vcmpud k2 {k1}, zmm1, S_i(zmm2/m_t), {le}
vpcmpnequd k2 {k1}, zmm1, S_i(zmm2/m_t)	vcmpud k2 {k1}, zmm1, S_i(zmm2/m_t), {neq}
vpcmpnltud k2 {k1}, zmm1, S_i(zmm2/m_t)	vcmpud k2 {k1}, zmm1, S_i(zmm2/m_t), {nlt}
vpcmpnleud k2 {k1}, zmm1, S_i(zmm2/m_t)	vcmpud k2 {k1}, zmm1, S_i(zmm2/m_t), {nle}

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

__mmask16	_mm512_cmp_epi32_mask(__m512i, __m512i, const _MM_CMPINT_ENUM);
__mmask16	_mm512_mask_cmp_epi32_mask(__mmask16, __m512i, __m512i, const _MM_CMPINT_ENUM);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPERMD - Permutes Int32 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W0 36 /r	vpermd zmm1 {k1}, zmm2, zmm3/mt	Element permute vector zmm3/mt using vector zmm2 as indices and store the final result in zmm1, under write-mask.

Description

Performs an element permutation of elements from int32 vector read from memory or vector zmm3, using int32 vector zmm2 element as source indices. The result of the permutation is written into int32 vector zmm1. Note that index values may be repeated so that the same vector source element can be replicated across many vector destination elements.

No swizzle, broadcast, or conversion is performed by this instruction.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
src[511:0] = zmm3/$m_t$

for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    j = zmm2[i+3:i]*32
    zmm1[i+31:i] = src[j+31:j]
  }
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i   _mm512_permutevar_epi32 (__m512i, __m512i);
__m512i   _mm512_mask_permutevar_epi32 (__m512i, __mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.
	This instruction does not support any SwizzUpConv different from the default value (no broadcast, no conversion). If SwizzUpConv function is set to any value different than "no action", then an Invalid Opcode fault is raised. This includes register swizzles.

VPERMF32X4 - Shuffle Vector Dqwords

Opcode	Instruction	Description
MVEX.512.66.0F3A.W0 07 /r ib	vpermf32x4 zmm1 {k1}, zmm2/m_t, imm8	4xFloat32 shuffle element vector zmm2/m_t and store the result in zmm1, using imm8 , under write-mask.

Description

Shuffles 128-bit blocks of the vector read from memory or vector zmm2/mem using index bits in immediate. The result of the shuffle is written into vector zmm1.

No swizzle, broadcast, or conversion is performed by this instruction.

This instruction is write-masked, 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.

Immediate Format

imm8	128-bit level permutation vector {3210}	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0

Operation

```
src[511:0] = zmm2/$m_t$


// Inter-lane shuffle
for (n = 0; n < 16/4; n++) {
  i = 128*n
  j = 128*((perm128 >> 2*n) & 0x3)
  tmp[i+127:i] = src[j+127:j]
}

// Writemasking
for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    zmm1[i+31:i] = tmp[i+31:i]
  }
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

__m512i	_mm512_permute4f128_epi32 (__m512i, _MM_PERM_ENUM);
__m512i	_mm512_mask_permute4f128_epi32 (__m512i, __mmask16, __m512i, _MM_PERM_ENUM);
__m512	_mm512_permute4f128_ps (__m512, _MM_PERM_ENUM);
__m512	_mm512_mask_permute4f128_ps (__m512, __mmask16, __m512, _MM_PERM_ENUM);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.
	This instruction does not support any SwizzUpConv different from the default value (no broadcast, no conversion). If SwizzUpConv function is set to any value different than "no action", then an Invalid Opcode fault is raised. This includes register swizzles.

VPGATHERDD - Gather Int32 Vector With Signed Dword Indices

Opcode	Instruction	Description
MVEX.512.66.0F38.W0 90 /r /vsib	vpgatherdd zmm1 {k1}, $U_{i32}(mv_t)$	Gather int32 vector $U_{i32}(mv_t)$ into int32 vector zmm1 using doubleword indices and k1 as completion mask.

Description

A set of 16 memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$ with scale $SCALE$ are converted to a int32 vector. The result is written into int32 vector zmm1.

Note the special mask behavior as only a subset of the active elements of write mask k1 are actually operated on (as denoted by function $SELECT\_SUBSET$). There are only two guarantees about the function: (a) the destination mask is a subset of the source mask (identity is included), and (b) on a given invocation of the instruction, **at least** one element (the least significant enabled mask bit) will be selected from the source mask.

Programmers should always enforce the execution of a gather/scatter instruction to be re-executed (via a loop) until the full completion of the sequence (i.e. all elements of the gather/scatter sequence have been loaded/stored and hence, the write-mask bits all are zero).

Note that accessed element by will always access 64 bytes of memory. The memory region accessed by each element will always be between *elemen_linear_address & (∼0x3F)* and *(element_linear_address & (∼0x3F)) + 63* boundaries.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Note also the special mask behavior as the corresponding bits in write mask k1 are reset with each destination element being updated according to the subset of write mask k1. This is useful to allow conditional re-trigger of the instruction until all the elements from a given write mask have been successfully loaded.

The instruction will #GP fault if the destination vector zmm1 is the same as index vector $VINDEX$.

Operation

```
// instruction works over a subset of the write mask
ktemp = SELECT_SUBSET(k1)

// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 16; n++) {
  if (ktemp[n] != 0) {
```

```
        i = 32*n
        // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[i+31:i] * SCALE)
        pointer[63:0] = $mv_t$[n]
        zmm1[i+31:i] = UpConvLoad$_{i32}$(pointer)
        k1[n] = 0
    }
}
```

Flags Affected

None.

Memory Up-conversion: U_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	1
101	sint8 to sint32	[rax] {sint8}	1
110	uint16 to uint32	[rax] {uint16}	2
111	sint16 to sint32	[rax] {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

__m512i	_mm512_i32gather_epi32 (__m512i, void const*, int);
__m512i	_mm512_mask_i32gather_epi32 (__m512i, __mmask16, __m512i, void const*, int);
__m512i	_mm512_i32extgather_epi32 (__m512i, void const*, _MM_UPCONV_EPI32_ENUM, int, int);
__m512i	_mm512_mask_i32extgather_epi32 (__m512i, __mmask16, __m512i, void const*, _MM_UPCONV_EPI32_ENUM, int, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is
	in a non-canonical form, and corresponding write-mask bit is not zero.
#GP(0)	If a memory address is in a non-canonical form,
	and corresponding write-mask bit is not zero.
	If a memory operand linear address is not aligned
	to element-wise data granularity dictated by the UpConv
	and corresponding write-mask bit is not zero.
	If the destination vector is the same as the index vector [see
	.
#PF(fault-code)	If a memory operand linear address produces a page fault
	and corresponding write-mask bit is not zero.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.
	If using a 16 bit effective address.
	If ModRM.rm is different than 100b.
	If no write mask is provided or selected write-mask is k0.

VPGATHERDQ - Gather Int64 Vector With Signed Dword Indices

Opcode	Instruction	Description
MVEX.512.66.0F38.W1 90 /r /vsib	vpgatherdq zmm1 {k1}, $U_{i64}(mv_t)$	Gather int64 vector $U_{i64}(mv_t)$ into int64 vector zmm1 using doubleword indices and k1 as completion mask.

Description

A set of 8 memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$ with scale $SCALE$ are converted to a int64 vector. The result is written into int64 vector zmm1.

Note the special mask behavior as only a subset of the active elements of write mask k1 are actually operated on (as denoted by function $SELECT\_SUBSET$). There are only two guarantees about the function: (a) the destination mask is a subset of the source mask (identity is included), and (b) on a given invocation of the instruction, **at least** one element (the least significant enabled mask bit) will be selected from the source mask.

Programmers should always enforce the execution of a gather/scatter instruction to be re-executed (via a loop) until the full completion of the sequence (i.e. all elements of the gather/scatter sequence have been loaded/stored and hence, the write-mask bits all are zero).

Note that accessed element by will always access 64 bytes of memory. The memory region accessed by each element will always be between *elemen_linear_address & (∼0x3F)* and *(element_linear_address & (∼0x3F)) + 63* boundaries.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Note also the special mask behavior as the corresponding bits in write mask k1 are reset with each destination element being updated according to the subset of write mask k1. This is useful to allow conditional re-trigger of the instruction until all the elements from a given write mask have been successfully loaded.

The instruction will #GP fault if the destination vector zmm1 is the same as index vector $VINDEX$.

Operation

```
// instruction works over a subset of the write mask
ktemp = SELECT_SUBSET(k1)

// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 8; n++) {
  if (ktemp[n] != 0) {
```

```
        i = 64*n
        j = 32*n
        // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[j+31:j] * SCALE)
        pointer[63:0] = $mv_t$[n]
        zmm1[i+63:i] = UpConvLoad$_{i64}$(pointer)
        k1[n] = 0
    }
  }
k1[15:8] = 0
```

Flags Affected

None.

Memory Up-conversion: U$_{i64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

__m512i	_mm512_i32logather_epi64 (__m512i, void const*, int);
__m512i	_mm512_mask_i32logather_epi64 (__m512i, __mmask8, __m512i, void const*, int);
__m512i	_mm512_i32loextgather_epi64 (__m512i, void const*, _MM_UPCONV_EPI64_ENUM, int, int);
__m512i	_mm512_mask_i32loextgather_epi64 (__m512i, __mmask8, __m512i, void const*, _MM_UPCONV_EPI64_ENUM, int, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form, and corresponding write-mask bit is not zero.
#GP(0)	If a memory address is in a non-canonical form, and corresponding write-mask bit is not zero. If a memory operand linear address is not aligned to element-wise data granularity dictated by the UpConv and corresponding write-mask bit is not zero. If the destination vector is the same as the index vector [see .
#PF(fault-code)	If a memory operand linear address produces a page fault and corresponding write-mask bit is not zero.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If using a 16 bit effective address. If ModRM.rm is different than 100b. If no write mask is provided or selected write-mask is k0.

VPMADD231D - Multiply First Source By Second Source and Add To Destination Int32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 B5 /r	vpmadd231d zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Multiply int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t), add the result to int32 vector zmm1, and store the final result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between int32 vector zmm2 and the int32 vector result of the swizzle/broadcast/conversion process on memory or vector int32 zmm3, then adds the result to int32 vector zmm1. The final sum is written into int32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // integer operation
    zmm1[i+31:i] = zmm2[i+31:i] * tmpSrc3[i+31:i] + zmm1[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i   _mm512_fmadd_epi32 (__m512i, __m512i, __m512i);
__m512i   _mm512_mask_fmadd_epi32 (__m512i, __mmask16, __m512i, __m512i);
__m512i   _mm512_mask3_fmadd_epi32 (__m512i, __m512i, __m512i, __mmask16);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPMADD233D - Multiply First Source By Specially Swizzled Second Source and Add To Second Source Int32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 B4 /r	vpmadd233d zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Multiply int32 vector zmm2 by certain elements of int32 vector S_{i32}(zmm3/m_t), add the result to certain elements of S_{i32}(zmm3/m_t), and store the final result in zmm1, under write-mask.

Description

This instruction is built around the concept of 4-element sets, of which there are four: elements 0-3, 4-7, 8-11, and 12-15. If we refer to the int32 vector result of the broadcast (no conversion is supported) process on memory or the int32 vector zmm3 (no swizzle is supported) as t3, then:

Each element 0-3 of int32 vector zmm2 is multiplied by element 1 of t3, the result is added to element 0 of t3, and the final sum is written into the corresponding element 0-3 of int32 vector zmm1.

Each element 4-7 of int32 vector zmm2 is multiplied by element 5 of t3, the result is added to element 4 of t3, and the final sum is written into the corresponding element 4-7 of int32 vector zmm1.

Each element 8-11 of int32 vector zmm2 is multiplied by element 9 of t3, the result is added to element 8 of t3, and the final sum is written into the corresponding element 8-11 of int32 vector zmm1.

Each element 12-15 of int32 vector zmm2 is multiplied by element 13 of t3, the result is added to element 12 of t3, and the final sum is written into the corresponding element 12-15 of int32 vector zmm1.

This instruction makes it possible to perform scale and bias in a single instruction without needing to have either scale or bias already loaded in a register. This saves one vector load for each interpolant, representing around ten percent of shader instructions.

For structure-of-arrays (SOA) operation, this instruction is intended to be used with the $\{4to16\}$ broadcast on src2, allowing all 16 scale and biases to be identical. For array-of-structures (AOS) vec4 operations, no broadcast is used, allowing four different scales and biases, one for each vec4.

No conversion or swizzling is supported for this instruction. However, all broadcasts except $\{1to16\}$ are supported (i.e. 16to16 and 4to16).

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    base = ( n & ~0x03 ) * 32
    scale[31:0] = tmpSrc3[base+63:base+32]
    bias[31:0] = tmpSrc3[base+31:base]
    // integer operation
    zmm1[i+31:i] = zmm2[i+31:i] * scale[31:0] + bias[31:0]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	reserved	N/A	N/A
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	reserved	N/A
010	reserved	N/A
011	reserved	N/A
100	reserved	N/A
101	reserved	N/A
110	reserved	N/A
111	reserved	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

__m512i	_mm512_fmadd233_epi32 (__m512i, __m512i);
__m512i	_mm512_mask_fmadd233_epi32 (__m512i, __mmask16, __m512i, __m512i);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to 16 or 64-byte (depending on the swizzle broadcast).
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. This instruction does not support any SwizzUpConv involving data conversion, register swizzling or $\{1to16\}$ broadcast. If SwizzUpConv function is set to any value different than "no action" or $\{4to16\}$ then

an Invalid Opcode fault is raised

VPMAXSD - Maximum of Int32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 3D /r	vpmaxsd zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Determine the maximum of int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Determines the maximum value of each pair of corresponding elements in int32 vector zmm2 and the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3. The result is written into int32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    zmm1[i+31:i] = IMax(zmm2[i+31:i] , tmpSrc3[i+31:i])
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_max_epi32 (__m512i, __m512i);
__m512i    _mm512_mask_max_epi32 (__m512i, __mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPMAXUD - Maximum of Uint32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 3F /r	vpmaxud zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Determine the maximum of uint32 vector zmm2 and uint32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Determines the maximum value of each pair of corresponding elements in uint32 vector zmm2 and the uint32 vector result of the swizzle/broadcast/conversion process on memory or uint32 vector zmm3. The result is written into uint32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // unsigned integer operation
    zmm1[i+31:i] = UMax(zmm2[i+31:i] , tmpSrc3[i+31:i])
  }
}
```

Flags Affected

None.

Memory Up-conversion: S$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_max_epu32 (__m512i,__m512i);
__m512i    _mm512_mask_max_epu32 (__m512i, __mmask16, __m512i,__m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPMINSD - Minimum of Int32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 39 /r	vpminsd zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Determine the minimum of int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Determines the minimum value of each pair of corresponding elements in int32 vector zmm2 and the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3. The result is written into int32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    zmm1[i+31:i] = (zmm2[i+31:i] < tmpSrc3[i+31:i]) ?
                zmm2[i+31:i] : tmpSrc3[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_min_epi32 (__m512i, __m512i);
__m512i    _mm512_mask_min_epi32 (__m512i, __mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPMINUD - Minimum of Uint32 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W0 3B /r	vpminud zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Determine the minimum of uint32 vector zmm2 and uint32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Determines the minimum value of each pair of corresponding elements in uint32 vector zmm2 and the uint32 vector result of the swizzle/broadcast/conversion process on memory or uint32 vector zmm3. The result is written into uint32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // unsigned integer operation
    zmm1[i+31:i] = UMin(zmm2[i+31:i] , tmpSrc3[i+31:i])
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_min_epu32 (__m512i, __m512i);
__m512i    _mm512_mask_min_epu32 (__m512i, __mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPMULHD - Multiply Int32 Vectors And Store High Result

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 87 /r	vpmulhd zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Multiply int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between int32 vector zmm2 and the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3. The high 32 bits of the result are written into int32 zmm1 vector.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    tmp[63:0] = zmm2[i+31:i] * tmpSrc3[i+31:i]
    zmm1[i+31:i] = tmp[63:32]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_mulhi_epi32 (__m512i, __m512i);
__m512i    _mm512_mask_mulhi_epi32 (__m512i, __mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPMULHUD - Multiply Uint32 Vectors And Store High Result

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W0 86 /r	vpmulhud zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Multiply uint32 vector zmm2 and uint32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between uint32 vector zmm2 and the uint32 vector result of the swizzle/broadcast/conversion process on memory or uint32 vector zmm3. The high 32 bits of the result are written into uint32 zmm1 vector.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // unsigned integer operation
    tmp[63:0] = zmm2[i+31:i] * tmpSrc3[i+31:i]
    zmm1[i+31:i] = tmp[63:32]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_mulhi_epu32 (__m512i, __m512i);
__m512i    _mm512_mask_mulhi_epu32 (__m512i, __mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPMULLD - Multiply Int32 Vectors And Store Low Result

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 40 /r	vpmulld zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Multiply int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element multiplication between int32 vector zmm2 and the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3, and the low 32 bits of the result are written into int32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    zmm1[i+31:i] = zmm2[i+31:i] * tmpSrc3[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_mullo_epi32 (__m512i, __m512i);
__m512i    _mm512_mask_mullo_epi32 (__m512i, __mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPORD - Bitwise OR Int32 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F.W0 EB /r	vpord zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Perform a bitwise OR between int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element bitwise OR between int32 vector zmm2 and the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3. The result is written into int32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] = zmm2[i+31:i] | tmpSrc3[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_or_epi32 (__m512i, __m512i);
__m512i    _mm512_mask_or_epi32 (__m512i, __mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPORQ - Bitwise OR Int64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F.W1 EB /r	vporq zmm1 {k1}, zmm2, S_{i64}(zmm3/m_t)	Perform a bitwise OR between int64 vector zmm2 and int64 vector S_{i64}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element bitwise OR between int64 vector zmm2 and the int64 vector result of the swizzle/broadcast/conversion process on memory or int64 vector zmm3. The result is written into int64 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    zmm1[i+63:i] = zmm2[i+63:i] | tmpSrc3[i+63:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S_{i64}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_or_epi64 (__m512i, __m512i);
__m512i    _mm512_mask_or_epi64 (__m512i, __mmask8, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPSBBD - Subtract Int32 Vectors with Borrow

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 5E /r	vpsbbd zmm1 {k1}, k2, S_{i32}(zmm3/m_t)	Subtract int32 vector S_{i32}(zmm3/m_t) and vector mask register k2 from int32 vector zmm1 and store the result in zmm1, and the borrow of the subtraction in k2, under write-mask.

Description

Performs an element-by-element three-input subtraction of the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3, as well as the corresponding bit of k2, from int32 vector zmm1. The result is written into int32 vector zmm1.

In addition, the borrow from the subtraction difference for the n-th element is written into the n-th bit of vector mask k2.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1 and k2. Elements in zmm1 and k2 with the corresponding bit clear in k1 retain their previous value.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // integer operation
    tmpBorrow = Borrow(zmm1[i+31:i] - k2[n] - tmpSrc3[i+31:i])
    zmm1[i+31:i] = zmm1[i+31:i] - k2[n] - tmpSrc3[i+31:i]
    k2[n] = tmpBorrow
  }
}
```

Flags Affected

None.

Memory Up-conversion: S$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

__m512i	_mm512_sbb_epi32 (__m512i, __mmask16, __m512i, __mmask16*);
__m512i	_mm512_mask_sbb_epi32 (__m512i, __mmask16, __mmask16, __m512i, __mmask16*);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPSBBRD - Reverse Subtract Int32 Vectors with Borrow

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 6E /r	vpsbbrd zmm1 {k1}, k2, S_{i32}(zmm3/m_t)	Subtract int32 vector zmm1 and vector mask register k2 from int32 vector S_{i32}(zmm3/m_t), and store the result in zmm1, and the borrow of the subtraction in k2, under write-mask.

Description

Performs an element-by-element three-input subtraction of int32 vector zmm1, as well as the corresponding bit of k2, from the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3. The result is written into int32 vector zmm1.

In addition, the borrow from the subtraction for the n-th element is written into the n-th bit of vector mask k2.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1 and k2. Elements in zmm1 and k2 with the corresponding bit clear in k1 retain their previous value.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // integer operation
    tmpBorrow = Borrow(tmpSrc3[i+31:i] - k2[n] - zmm1[i+31:i])
    zmm1[i+31:i] = tmpSrc3[i+31:i] - k2[n] - zmm1[i+31:i]
    k2[n] = tmpBorrow
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_sbbr_epi32 (__m512i, __mmask16, __m512i, __mmask16*);
__m512i    _mm512_mask_sbbr_epi32 (__m512i, __mmask16, __mmask16, __m512i,
           __mmask16*);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPSCATTERDD - Scatter Int32 Vector With Signed Dword Indices

Opcode	**Instruction**	**Description**
MVEX.512.66.0F38.W0 A0 /r /vsib	vpscatterdd mv_t {k1}, D_{i32}(zmm1)	Scatter int32 vector D_{i32}(zmm1) to vector memory locations mv_t using doubleword indices and k1 as completion mask.

Description

Down-converts and stores all 16 elements in int32 vector UNDEF to the memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$, with scale $SCALE$.

Note the special mask behavior as only a subset of the active elements of write mask k1 are actually operated on (as denoted by function $SELECT\_SUBSET$). There are only two guarantees about the function: (a) the destination mask is a subset of the source mask (identity is included), and (b) on a given invocation of the instruction, **at least** one element (the least significant enabled mask bit) will be selected from the source mask.

Programmers should always enforce the execution of a gather/scatter instruction to be re-executed (via a loop) until the full completion of the sequence (i.e. all elements of the gather/scatter sequence have been loaded/stored and hence, the write-mask bits all are zero).

Writes to overlapping destination memory locations are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). 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.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element after down-conversion.

Note also the special mask behavior as the corresponding bits in write mask k1 are reset with each destination element being updated according to the subset of write mask k1. This is useful to allow conditional re-trigger of the instruction until all the elements from a given write mask have been successfully stored.

Operation

```
// instruction works over a subset of the write mask
ktemp = SELECT_SUBSET(k1)

// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 16; n++) {
```

```
        if (ktemp[n] != 0) {
          i = 32*n
          // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[i+31:i] * SCALE)
          pointer[63:0] = $mv_t$[n]
          tmp = DownConvStore$_{i32}$(UNDEF[i+31:i], SSS[2:0])
          if(DownConvStoreSizeOf$_{i32}$(SSS[2:0]) == 4) {
            MemStore(pointer) = tmp[31:0]
          } else if(DownConvStoreSizeOf$_{i32}$(SSS[2:0]) == 2) {
            MemStore(pointer) = tmp[15:0]
          } else if(DownConvStoreSizeOf$_{i32}$(SSS[2:0]) == 1) {
            MemStore(pointer) = tmp[7:0]
          }
          k1[n] = 0
        }
      }
```

Flags Affected

None.

Memory Down-conversion: D$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	uint32 to uint8	zmm1 {uint8}	1
101	sint32 to sint8	zmm1 {sint8}	1
110	uint32 to uint16	zmm1 {uint16}	2
111	sint32 to sint16	zmm1 {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

void	_mm512_i32scatter_epi32 (void*, __m512i, __m512i, int);
void	_mm512_mask_i32scatter_epi32 (void*, __mmask16, __m512i, __m512i, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form, and corresponding write-mask bit is not zero.
#GP(0)	If a memory address is in a non-canonical form, and corresponding write-mask bit is not zero. If a memory operand linear address is not aligned to element-wise data granularity dictated by the DownConv mode, and corresponding write-mask bit is not zero.
#PF(fault-code)	If a memory operand linear address produces a page fault and corresponding write-mask bit is not zero.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If using a 16 bit effective address. If ModRM.rm is different than 100b. If no write mask is provided or selected write-mask is k0.

VPSCATTERDQ - Scatter Int64 Vector With Signed Dword Indices

Opcode	Instruction			Description
MVEX.512.66.0F38.W1 A0 /r /vsib	vpscatterdq D_{i64}(zmm1)	mv_t	{k1},	Scatter int64 vector D_{i64}(zmm1) to vector memory locations mv_t using doubleword indices and k1 as completion mask.

Description

Down-converts and stores all 8 elements in int64 vector UNDEF to the memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$, with scale $SCALE$.

Note the special mask behavior as only a subset of the active elements of write mask k1 are actually operated on (as denoted by function $SELECT\_SUBSET$). There are only two guarantees about the function: (a) the destination mask is a subset of the source mask (identity is included), and (b) on a given invocation of the instruction, **at least** one element (the least significant enabled mask bit) will be selected from the source mask.

Programmers should always enforce the execution of a gather/scatter instruction to be re-executed (via a loop) until the full completion of the sequence (i.e. all elements of the gather/scatter sequence have been loaded/stored and hence, the write-mask bits all are zero).

Writes to overlapping destination memory locations are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). 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.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element after down-conversion.

Note also the special mask behavior as the corresponding bits in write mask k1 are reset with each destination element being updated according to the subset of write mask k1. This is useful to allow conditional re-trigger of the instruction until all the elements from a given write mask have been successfully stored.

Operation

```
// instruction works over a subset of the write mask
ktemp = SELECT_SUBSET(k1)

// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 8; n++) {
```

```
    if (ktemp[n] != 0) {
      i = 64*n
      j = 32*n
      // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[j+31:j] * SCALE)
      pointer[63:0] = $mv_t$[n]
      tmp = DownConvStore$_{i64}$(UNDEF[i+63:i], SSS[2:0])
      if(DownConvStoreSizeOf$_{i64}$(SSS[2:0]) == 8) {
        MemStore(pointer) = tmp[63:0]
      }
      k1[n] = 0
    }
  }
  k1[15:8] = 0
```

Flags Affected

None.

Memory Down-conversion: D$_{i64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

void	_mm512_i32loscatter_epi64 (void*, __m512i, __m512i, int);
void	_mm512_mask_i32loscatter_epi64 (void*, __mmask8, __m512i, __m512i, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form, and corresponding write-mask bit is not zero.
#GP(0)	If a memory address is in a non-canonical form, and corresponding write-mask bit is not zero. If a memory operand linear address is not aligned to element-wise data granularity dictated by the DownConv mode, and corresponding write-mask bit is not zero.
#PF(fault-code)	If a memory operand linear address produces a page fault and corresponding write-mask bit is not zero.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If using a 16 bit effective address. If ModRM.rm is different than 100b. If no write mask is provided or selected write-mask is k0.

VPSHUFD - Shuffle Vector Doublewords

Opcode	Instruction	Description
MVEX.512.66.0F.W0 70 /r ib	vpshufd zmm1 {k1}, zmm2/m_t, imm8	Dword shuffle int32 vector zmm2/m_t and store the result in zmm1, using imm8 , under write-mask.

Description

Shuffles 32 bit blocks of the vector read from memory or vector zmm2/mem using index bits in immediate. The result of the shuffle is written into vector zmm1.

No swizzle, broadcast, or conversion is performed by this instruction.

This instruction is write-masked, 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.

Immediate Format

imm8	32 bit level permutation vector {dcba}	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0

Operation

```
src[511:0] = zmm2/$m_t$


// Intra-lane shuffle
for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    // offset within 128-bit chunk
    j = 32*((perm32 >> 2*(n & 0x3)) & 0x3)
    // 128-bit level offset
    j = j + 128*(n >> 2)
    zmm1[i+31:i] = src[j+31:j]
  }
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

__m512i	_mm512_shuffle_epi32 (__m512i, _MM_PERM_ENUM);
__m512i	_mm512_mask_shuffle_epi32 (__m512i, __mmask16, __m512i, _MM_PERM_ENUM);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.
	This instruction does not support any SwizzUpConv different from the default value (no broadcast, no conversion). If SwizzUpConv function is set to any value different than "no action", then an Invalid Opcode fault is raised. This includes register swizzles.

VPSLLD - Shift Int32 Vector Immediate Left Logical

Opcode	Instruction	Description
MVEX.NDD.512.66.0F.W0 72 /6 ib	vpslld zmm1 {k1}, S_{i32}(zmm2/m_t), $imm8$	Shift left int32 vector S_{i32}(zmm2/m_t) and store the result in zmm1, using $imm8$, under write-mask.

Description

Performs an element-by-element logical left shift of the result of the swizzle/broadcast/conversion process on memory or vector int32 zmm2, shifting by the number of bits specified in immediate field. The result is stored in int32 vector zmm1.

If the value specified by the shift operand is greater than 31 then the destination operand is set to all 0s.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{i32}$(zmm2/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // integer operation
    zmm1[i+31:i] = tmpSrc2[i+31:i] << IMM8[7:0]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i   _mm512_slli_epi32 (__m512i, unsigned int);
__m512i   _mm512_mask_slli_epi32 (__m512i, __mmask16, __m512i, unsigned int);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPSLLVD - Shift Int32 Vector Left Logical

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W0 47 /r	vpsllvd zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Shift left int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element left shift of int32 vector zmm2, shifting by the number of bits specified by the int32 vector result of the swizzle/broadcast/conversion process on memory or vector int32 zmm3. The result is stored in int32 vector zmm1.

If the value specified by the shift operand is greater than 31 then the destination operand is set to all 0s.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    zmm1[i+31:i] = zmm2[i+31:i] << tmpSrc3[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_sllv_epi32 (__m512i, __m512i);
__m512i    _mm512_mask_sllv_epi32 (__m512i, __mmask16, __m512i,__m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPSRAD - Shift Int32 Vector Immediate Right Arithmetic

Opcode	Instruction	Description
MVEX.NDD.512.66.0F.W0 72 /4 ib	vpsrad zmm1 {k1}, S_{i32}(zmm2/m_t), $imm8$	Shift right arithmetic int32 vector S_{i32}(zmm2/m_t) and store the result in zmm1, using $imm8$, under write-mask.

Description

Performs an element-by-element arithmetic right shift of the result of the swizzle/broadcast/conversion process on memory or vector int32 zmm2, shifting by the number of bits specified in immediate field. The result is stored in int32 vector zmm1.

An arithmetic right shift leaves the sign bit unchanged after each shift count, so the final result has the i+1 msbs set to the original sign bit, where i is the number of bits by which to shift right.

If the value specified by the shift operand is greater than 31 each destination data element is filled with the initial value of the sign bit of the element.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{i32}$(zmm2/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    zmm1[i+31:i] = tmpSrc2[i+31:i] >> IMM8[7:0]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_srai_epi32 (__m512i, unsigned int);
__m512i    _mm512_mask_srai_epi32 (__m512i, __mmask16, __m512i, unsigned int);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPSRAVD - Shift Int32 Vector Right Arithmetic

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W0 46 /r	vpsravd zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Shift right arithmetic int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element arithmetic right shift of int32 vector zmm2, shifting by the number of bits specified by the int32 vector result of the swizzle/broadcast/conversion process on memory or vector int32 zmm3. The result is stored in int32 vector zmm1.

An arithmetic right shift leaves the sign bit unchanged after each shift count, so the final result has the i+1 msbs set to the original sign bit, where i is the number of bits by which to shift right.

If the value specified by the shift operand is greater than 31 each destination data element is filled with the initial value of the sign bit of the element.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    zmm1[i+31:i] = zmm2[i+31:i] >> tmpSrc3[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_srav_epi32 (__m512i, __m512i);
__m512i    _mm512_mask_srav_epi32 (__m512i, __mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPSRLD - Shift Int32 Vector Immediate Right Logical

Opcode	Instruction	Description
MVEX.NDD.512.66.0F.W0 72 /2 ib	vpsrld zmm1 {k1}, S_{i32}(zmm2/m_t), $imm8$	Shift right logical int32 vector S_{i32}(zmm2/m_t) and store the result in zmm1, using $imm8$, under write-mask.

Description

Performs an element-by-element logical right shift of the result of the swizzle/broadcast/conversion process on memory or vector int32 zmm2, shifting by the number of bits specified in immediate field. The result is stored in int32 vector zmm1.

A logical right shift shifts a 0-bit into the msb for each shift count, so the final result has the i msbs set to 0, where i is the number of bits by which to shift right.

If the value specified by the shift operand is greater than 31 then the destination operand is set to all 0s.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{i32}$(zmm2/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    zmm1[i+31:i] = tmpSrc2[i+31:i] >> IMM8[7:0]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_srli_epi32 (__m512i, unsigned int);
__m512i    _mm512_mask_srli_epi32 (__m512i, __mmask16, __m512i, unsigned int);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPSRLVD - Shift Int32 Vector Right Logical

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W0 45 /r	vpsrlvd zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Shift right logical int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element logical right shift of int32 vector zmm2, shifting by the number of bits specified by the int32 vector result of the swizzle/broadcast/conversion process on memory or vector int32 zmm3. The result is stored in int32 vector zmm1.

A logical right shift shifts a 0-bit into the msb for each shift count, so the final result has the i msbs set to 0, where i is the number of bits by which to shift right.

If the value specified by the shift operand is greater than 31 then the destination operand is set to all 0s.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    zmm1[i+31:i] = zmm2[i+31:i] >> tmpSrc3[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_srlv_epi32 (__m512i, __m512i);
__m512i    _mm512_mask_srlv_epi32 (__m512i, __mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPSUBD - Subtract Int32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F.W0 FA /r	vpsubd zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Subtract int32 vector S_{i32}(zmm3/m_t) from int32 vector zmm2 and store the result in zmm1, under write-mask.

Description

Performs an element-by-element subtraction from int32 vector zmm2 of the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3. The result is written into int32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // integer operation
    zmm1[i+31:i] = zmm2[i+31:i] - tmpSrc3[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_sub_epi32 (__m512i, __m512i);
__m512i    _mm512_mask_sub_epi32 (__m512i, __mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPSUBRD - Reverse Subtract Int32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 6C /r	vpsubrd zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Subtract int32 vector zmm2 from int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element subtraction of int32 vector zmm2 from the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3. The result is written into int32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // integer operation
    zmm1[i+31:i] = -zmm2[i+31:i] + tmpSrc3[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_subr_epi32 (__m512i, __m512i);
__m512i    _mm512_mask_subr_epi32 (__m512i, __mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPSUBRSETBD - Reverse Subtract Int32 Vectors and Set Borrow

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 6F /r	vpsubrsetbd zmm1 {k1}, k2, S_{i32}(zmm3/m_t)	Subtract int32 vector zmm1 from int32 vector S_{i32}(zmm3/m_t) and store the subtraction in zmm1 and the borrow from the subtraction in k2, under write-mask.

Description

Performs an element-by-element subtraction of int32 vector zmm1 from the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3. The result is written into int32 vector zmm1.

In addition, the borrow from the subtraction for the n-th element is written into the n-th bit of vector mask k2.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1 and k2. Elements in zmm1 and k2 with the corresponding bit clear in k1 retain their previous value.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // integer operation
    k2[n] = Borrow(tmpSrc3[i+31:i] - zmm1[i+31:i])
    zmm1[i+31:i] = tmpSrc3[i+31:i] - zmm1[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i   _mm512_subrsetb_epi32 (__m512i, __m512i, __mmask16*);
__m512i   _mm512_mask_subrsetb_epi32 (__m512i, __mmask16, __mmask16, __m512i,
          __mmask16*);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPSUBSETBD - Subtract Int32 Vectors and Set Borrow

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F38.W0 5F /r	vpsubsetbd zmm1 {k1}, k2, S_{i32}(zmm3/m_t)	Subtract int32 vector S_{i32}(zmm3/m_t) from int32 vector zmm1 and store the subtraction in zmm1 and the borrow from the subtraction in k2, under write-mask.

Description

Performs an element-by-element subtraction of the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3 from int32 vector zmm1. The result is written into int32 vector zmm1.

In addition, the borrow from the subtraction for the n-th element is written into the n-th bit of vector mask k2.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1 and k2. Elements in zmm1 and k2 with the corresponding bit clear in k1 retain their previous value.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // integer operation
    k2[n] = Borrow(zmm1[i+31:i] - tmpSrc3[i+31:i])
    zmm1[i+31:i] = zmm1[i+31:i] - tmpSrc3[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S_{i32}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

__m512i	_mm512_subsetb_epi32 (__m512i, __m512i, __mmask16*);
__m512i	_mm512_mask_subsetb_epi32 (__m512i, __mmask16, __mmask16, __m512i, __mmask16*);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPTESTMD - Logical AND Int32 Vectors and Set Vector Mask

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 27 /r	vptestmd k2 {k1}, zmm1, S_{i32}(zmm2/m_t)	Perform a bitwise AND between int32 vector zmm1 and int32 vector S_{i32}(zmm2/m_t), and set vector mask k2 to reflect the zero/non-zero status of each element of the result, under write-mask.

Description

Performs an element-by-element bitwise AND between int32 vector zmm1 and the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm2, and uses the result to construct a 16 bit vector mask, with a 0-bit for each element for which the result of the AND was 0, and a 1-bit where the result of the AND was not zero. The final result is written into vector mask k2.

The write-mask does not perform the normal write-masking function for this instruction. While it does enable/disable comparisons, it does not block updating of the destination; instead, if a write-mask bit is 0, the corresponding destination bit is set to 0. Nonetheless, the operation is similar enough so that it makes sense to use the usual write-mask notation. This mode of operation is desirable because the result will be used directly as a write-mask, rather than the normal case where the result is used with a separate write-mask that keeps the masked elements inactive.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{i32}$(zmm2/$m_t$)
}

for (n = 0; n < 16; n++) {
  k2[n] = 0
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    if ((zmm1[i+31:i] & tmpSrc2[i+31:i]) != 0) {
      k2[n] = 1
    }
  }
}
```

Flags Affected

None.

Memory Up-conversion: S$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__mmask16    _mm512_test_epi32_mask (__m512i, __m512i);
__mmask16    _mm512_mask_test_epi32_mask (__mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPXORD - Bitwise XOR Int32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F.W0 EF /r	vpxord zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Perform a bitwise XOR between int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element bitwise XOR between int32 vector zmm2 and the int32 vector result of the swizzle/broadcast/conversion process on memory or int32 vector zmm3. The result is written into int32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // signed integer operation
    zmm1[i+31:i] = zmm2[i+31:i] ^ tmpSrc3[i+31:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_xor_epi32 (__m512i, __m512i);
__m512i    _mm512_mask_xor_epi32 (__m512i, __mmask16, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VPXORQ - Bitwise XOR Int64 Vectors

Opcode	**Instruction**	**Description**
MVEX.NDS.512.66.0F.W1 EF /r	vpxorq zmm1 {k1}, zmm2, S_{i64}(zmm3/m_t)	Perform a bitwise XOR between int64 vector zmm2 and int64 vector S_{i64}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element bitwise XOR between int64 vector zmm2 and the int64 vector result of the swizzle/broadcast/conversion process on memory or int64 vector zmm3. The result is written into int64 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   tmpSrc3[511:0] = SwizzUpConvLoad$_{i64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    zmm1[i+63:i] = zmm2[i+63:i] ^ tmpSrc3[i+63:i]
  }
}
```

Flags Affected

None.

Memory Up-conversion: S_{i64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S_{i64}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512i    _mm512_xor_epi64 (__m512i, __m512i);
__m512i    _mm512_mask_xor_epi64 (__m512i, __mmask8, __m512i, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VRCP23PS - Reciprocal of Float32 Vector

Opcode	Instruction	Description
MVEX.512.66.0F38.W0 CA /r	vrcp23ps zmm1 {k1}, zmm2/m_t	Compute the approximate reciprocals float32 vector zmm2/m_t and store the result in zmm1, under write-mask.

Description

Computes the element-by-element reciprocal approximation of the float32 vector on memory or float32 vector zmm2 with 0.912ULP (relative error). The result is written into float32 vector zmm1.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is $\pm\infty$, 0.0 is returned for that element. Also, if any source element is ± 0.0, $\pm\infty$ is returned for that element.

Current implementation of this instruction does not support any SwizzUpConv setting other than "no broadcast and no conversion"; any other SwizzUpConv setting will result in an Invalid Opcode exception.

recip_1ulp() function follows Table 6.26 when dealing with floating-point special number.

Input	Result	Comment
NaN	input qNaN	raise #I flag if sNaN
$+\infty$	$+0$	
+0	$+\infty$	raise #Z flag
-0	$-\infty$	raise #Z flag
$-\infty$	-0	
2^n	2^{-n}	exact result

Table 6.26: recip_1ulp() special floating-point values behavior

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
tmpSrc2[511:0] = zmm2/$m_t$

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags()    // SAE
}

for (n = 0; n < 16; n++) {
```

```
    if (k1[n] != 0) {
      i = 32*n
      zmm1[i+31:i] = recip_1ulp(tmpSrc2[i+31:i])
    }
  }
```

SIMD Floating-Point Exceptions

Invalid, Zero.

Denormal Handling

Treat Input Denormals As Zeros :
`YES`

Flush Tiny Results To Zero :
`YES`

Register Swizzle

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	reserved	N/A
010	reserved	N/A
011	reserved	N/A
100	reserved	N/A
101	reserved	N/A
110	reserved	N/A
111	reserved	N/A

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

__m512	_mm512_rcp23_ps (__m512);
__m512	_mm512_mask_rcp23_ps (__m512, __mmask16, __m512);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.
	This instruction does not support any SwizzUpConv different from the default value (no broadcast, no conversion). If SwizzUpConv function is set to any value different than "no action", then an Invalid Opcode fault is raised. This includes register swizzles.

VRNDFXPNTPD - Round Float64 Vector

Opcode	**Instruction**	**Description**
MVEX.512.66.0F3A.W1 52 /r ib	vrndfxpntpd zmm1 {k1}, S_{f64}(zmm2/m_t), $imm8$	Round float64 vector S_{f64}(zmm2/m_t) and store the result in zmm1, using $imm8$, under write-mask.

Description

Performs an element-by-element rounding of the result of the swizzle/broadcast/conversion from memory or float64 vector zmm2. The rounding result for each element is a float64 containing an integer or fixed-point value, depending on the value of expadj; the direction of rounding depends on the value of RC. The result is written into float64 vector zmm1.

This instruction doesn't actually convert the result to an int64; the results are float64s, just like the input, but are float64s containing the integer or fixed-point values that result from the specified rounding and scaling.

RoundToInt() function follows Table 6.27 when dealing with floating-point special number.

Input	Result
NaN	quietized input NaN
$+\infty$	$+\infty$
+0	$+0$
-0	-0
$-\infty$	$-\infty$

Table 6.27: RoundToInt() special floating-point values behavior

This instruction is write-masked, 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.

Immediate Format

	Rounding Mode	I_1	I_0
rn	Round to Nearest (even)	0	0
rd	Round Down (Round toward Negative Infinity)	0	1
ru	Round Up (Round toward Positive Infinity)	1	0
rz	Round toward Zero	1	1

Exponent Adjustment	value	I_7	I_6	I_5	I_4
0	2^0 (64.0 - no exponent adjustment)	0	0	0	0
4	2^4 (60.4)	0	0	0	1
5	2^5 (59.5)	0	0	1	0
8	2^8 (56.8)	0	0	1	1
16	2^{16} (48.16)	0	1	0	0
24	2^{24} (40.24)	0	1	0	1
31	2^{31} (33.31)	0	1	1	0
32	2^{32} (32.32)	0	1	1	1
reserved	*must UD*	1	x	x	x

Operation

```
RoundingMode = IMM8[1:0]
expadj = IMM8[6:4]

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{f64}$(zmm2/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] =
      RoundToInt(tmpSrc2[i+63:i] * EXPADJ_TABLE[expadj], RoundingMode) /
      EXPADJ_TABLE[expadj]
  }
}
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Denormal Handling

Treat Input Denormals As Zeros :

```
        (MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
    NO
```

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

__m512d _mm512_roundfxpnt_adjust_pd (__m512d, int, _MM_EXP_ADJ_ENUM);

__m512d _mm512_mask_roundfxpnt_adjust_pd (__m512d, __mmask8, __m512d, int , _MM_EXP_ADJ_ENUM);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VRNDFXPNTPS - Round Float32 Vector

Opcode	**Instruction**	**Description**
MVEX.512.66.0F3A.W0 52 /r ib	vrndfxpntps zmm1 {k1}, S_{f32}(zmm2/m_t), $imm8$	Round float32 vector S_{f32}(zmm2/m_t) and store the result in zmm1, using $imm8$, under write-mask.

Description

Performs an element-by-element rounding of the result of the swizzle/broadcast/conversion from memory or float32 vector zmm2. The rounding result for each element is a float32 containing an integer or fixed-point value, depending on the value of expadj; the direction of rounding depends on the value of RC. The result is written into float32 vector zmm1.

This instruction doesn't actually convert the result to an int32; the results are float32s, just like the input, but are float32s containing the integer or fixed-point values that result from the specified rounding and scaling.

RoundToInt() function follows Table 6.28 when dealing with floating-point special number.

This instruction treats input denormals as zeros according to the DAZ control bit, but does not flush tiny results to zero.

Input	Result
NaN	quietized input NaN
$+\infty$	$+\infty$
+0	$+0$
-0	-0
$-\infty$	$-\infty$

Table 6.28: RoundToInt() special floating-point values behavior

This instruction is write-masked, 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.

Immediate Format

	Rounding Mode	I_1	I_0
rn	Round to Nearest (even)	0	0
rd	Round Down (Round toward Negative Infinity)	0	1
ru	Round Up (Round toward Positive Infinity)	1	0
rz	Round toward Zero	1	1

Exponent Adjustment	value	I_7	I_6	I_5	I_4
0	2^0 (32.0 - no exponent adjustment)	0	0	0	0
4	2^4 (28.4)	0	0	0	1
5	2^5 (27.5)	0	0	1	0
8	2^8 (24.8)	0	0	1	1
16	2^{16} (16.16)	0	1	0	0
24	2^{24} (8.24)	0	1	0	1
31	2^{31} (1.31)	0	1	1	0
32	2^{32} (0.32)	0	1	1	1
reserved	*must UD*	1	x	x	x

Operation

```
RoundingMode = IMM8[1:0]
expadj = IMM8[6:4]

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   tmpSrc2[511:0] = zmm2[511:0]
} else {
   tmpSrc2[511:0] = SwizzUpConvLoad$_{f32}$(zmm2/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] =
      RoundToInt(tmpSrc2[i+31:i] * EXPADJ_TABLE[expadj], RoundingMode) /
      EXPADJ_TABLE[expadj]
  }
}
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Denormal Handling

Treat Input Denormals As Zeros :
`(MXCSR.DAZ)? YES : NO`

Flush Tiny Results To Zero :
`NO`

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_roundfxpnt_adjust_ps (__m512, int , _MM_EXP_ADJ_ENUM);
__m512    _mm512_mask_roundfxpnt_adjust_ps (__m512, __mmask16, __m512, int ,
          _MM_EXP_ADJ_ENUM);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VRSQRT23PS - Vector Reciprocal Square Root of Float32 Vector

Opcode	**Instruction**	**Description**
MVEX.512.66.0F38.W0 CB /r	vrsqrt23ps zmm1 {k1}, zmm2/m_t	Reciprocal square root float32 vector zmm2/m_t and store the result in zmm1, under write-mask.

Description

Computes the element-by-element reciprocal square root of the float32 vector on memory or float32 vector zmm2 with a precision of 0.775ULP (relative error). The result is written into float32 vector zmm1.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative source numbers, as well as $-\infty$, return the canonical NaN and set the Invalid Flag (#I).

Current implementation of this instruction does not support any SwizzUpConv setting other than "no broadcast and no conversion"; any other SwizzUpConv setting will result in an Invalid Opcode exception.

rsqrt_1ulp() function follows Table 6.29 when dealing with floating-point special number.

For an input value of $+/-0$ the instruction returns $-\infty$ and sets the Divide-By-Zero flag (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that this instruction treats input denormals as zeros of the same sign, so for denormal negative inputs it returns $-\infty$ and sets the Divide-By-Zero status flag.

Input	Result	Comments
NaN	input qNaN	Raise #I flag if sNaN
$+\infty$	$+0$	
$+0$	$+\infty$	Raise #Z flag
-0	$-\infty$	Raise #Z flag
<0	NaN	Raise #I flag
$-\infty$	NaN	Raise #I flag
2^{2n}	2^{-n}	exact result

Table 6.29: rsqrt_1ulp() special floating-point values behavior

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
tmpSrc2[511:0] = zmm2/$m_t$

if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags()    // SAE
}

for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    zmm1[i+31:i] = rsqrt_1ulp(tmpSrc2[i+31:i])
  }
}
```

SIMD Floating-Point Exceptions

Invalid, Zero.

Denormal Handling

Treat Input Denormals As Zeros :
`YES`

Flush Tiny Results To Zero :
`YES`

Register Swizzle

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	reserved	N/A
010	reserved	N/A
011	reserved	N/A
100	reserved	N/A
101	reserved	N/A
110	reserved	N/A
111	reserved	N/A

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
1xx	SAE (Supress-All-Exceptions)	, {sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    __ICL_INTRINCC _mm512_rsqrt23_ps (__m512);
__m512    __ICL_INTRINCC _mm512_mask_rsqrt23_ps (__m512, __mmask16, __m512);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1.
	If preceded by any REX, F0, F2, F3, or 66 prefixes.
	This instruction does not support any SwizzUpConv different from the default value (no broadcast, no conversion). If SwizzUpConv function is set to any value different than "no action", then an Invalid Opcode fault is raised. This includes register swizzles.

VSCALEPS - Scale Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 84 /r	vscaleps zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Multiply float32 vector zmm2 by 2 raised to the int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element scale of float32 vector zmm2 by multiplying it by 2^{exp}, where exp is the vector int32 result of the swizzle/broadcast/conversion process on memory or vector int32 zmm3. The result is written into vector float32 zmm1.

This instruction is needed for scaling u and v coordinates according to the mipmap size, which is 2^{mipmap_level}, and for the evaluation of Exp2.

Cases where the exponent would go out of range are handled as if multiplication (via vmulps) of zmm2 by 2^{zmm3} had been performed.

If the result cannot be represented with a float32, then the properly signed ∞ (for positive scaling operand) or 0 (for negative scaling operand) will be returned.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoadI(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    exp[31:0] = tmpSrc3[i+31:i]
    // signed int scale operation. float32 multiplication
    zmm1[i+31:i] = zmm2[i+31:i] * $2^{exp[31:0]}$
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :

```
(MXCSR.FZ)? YES : NO
```

Memory Up-conversion: S$_{i32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	reserved	N/A	N/A
100	uint8 to uint32	[rax] {uint8}	16
101	sint8 to sint32	[rax] {sint8}	16
110	uint16 to uint32	[rax] {uint16}	32
111	sint16 to sint32	[rax] {sint16}	32

Register Swizzle: S$_{i32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_scale_ps (__m512, __m512i);
__m512    _mm512_mask_scale_ps (__m512, __mmask16, __m512, __m512i);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.

#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VSCATTERDPD - Scatter Float64 Vector With Signed Dword Indices

Opcode	Instruction	Description
MVEX.512.66.0F38.W1 A2 /r /vsib	vscatterdpd mv_t {k1}, D_{f64}(zmm1)	Scatter float64 vector D_{f64}(zmm1) to vector memory locations mv_t using doubleword indices and k1 as completion mask.

Description

Down-converts and stores all 8 elements in float64 vector zmm1 to the memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$, with scale $SCALE$.

Note the special mask behavior as only a subset of the active elements of write mask k1 are actually operated on (as denoted by function $SELECT\_SUBSET$). There are only two guarantees about the function: (a) the destination mask is a subset of the source mask (identity is included), and (b) on a given invocation of the instruction, **at least** one element (the least significant enabled mask bit) will be selected from the source mask.

Programmers should always enforce the execution of a gather/scatter instruction to be re-executed (via a loop) until the full completion of the sequence (i.e. all elements of the gather/scatter sequence have been loaded/stored and hence, the write-mask bits all are zero).

Writes to overlapping destination memory locations are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). 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.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element after down-conversion.

Note also the special mask behavior as the corresponding bits in write mask k1 are reset with each destination element being updated according to the subset of write mask k1. This is useful to allow conditional re-trigger of the instruction until all the elements from a given write mask have been successfully stored.

Operation

```
// instruction works over a subset of the write mask
ktemp = SELECT_SUBSET(k1)

// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 8; n++) {
```

```
    if (ktemp[n] != 0) {
      i = 64*n
      j = 32*n
      // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[j+31:j] * SCALE)
      pointer[63:0] = $mv_t$[n]
      tmp = DownConvStore$_{f64}$(zmm1[i+63:i], SSS[2:0])
      if(DownConvStoreSizeOf$_{f64}$(SSS[2:0]) == 8) {
        MemStore(pointer) = tmp[63:0]
      }
      k1[n] = 0
    }
  }
  k1[15:8] = 0
```

SIMD Floating-Point Exceptions

None.

Memory Down-conversion: D$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm512_i32loextscatter_pd (void*, __m512i, __m512d, _MM_DOWNCONV_PD_ENUM, int, int);
void _mm512_mask_i32loextscatter_pd (void*, __mmask8, __m512i, __m512d, _MM_DOWNCONV_PD_ENUM, int, int);
void _mm512_i32loscatter_pd (void*, __m512i, __m512d, int);
void _mm512_mask_i32loscatter_pd (void*, __mmask8, __m512i, __m512d, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form, and corresponding write-mask bit is not zero.
#GP(0)	If a memory address is in a non-canonical form, and corresponding write-mask bit is not zero. If a memory operand linear address is not aligned to element-wise data granularity dictated by the DownConv mode, and corresponding write-mask bit is not zero.
#PF(fault-code)	If a memory operand linear address produces a page fault and corresponding write-mask bit is not zero.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If using a 16 bit effective address. If ModRM.rm is different than 100b. If no write mask is provided or selected write-mask is k0.

VSCATTERDPS - Scatter Float32 Vector With Signed Dword Indices

Opcode	Instruction	Description
MVEX.512.66.0F38.W0 A2 /r /vsib	vscatterdps mv_t {k1}, D_{f32}(zmm1)	Scatter float32 vector D_{f32}(zmm1) to vector memory locations mv_t using doubleword indices and k1 as completion mask.

Description

Down-converts and stores all 16 elements in float32 vector zmm1 to the memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$, with scale $SCALE$.

Note the special mask behavior as only a subset of the active elements of write mask k1 are actually operated on (as denoted by function $SELECT\_SUBSET$). There are only two guarantees about the function: (a) the destination mask is a subset of the source mask (identity is included), and (b) on a given invocation of the instruction, **at least** one element (the least significant enabled mask bit) will be selected from the source mask.

Programmers should always enforce the execution of a gather/scatter instruction to be re-executed (via a loop) until the full completion of the sequence (i.e. all elements of the gather/scatter sequence have been loaded/stored and hence, the write-mask bits all are zero).

Writes to overlapping destination memory locations are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). 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.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element after down-conversion.

Note also the special mask behavior as the corresponding bits in write mask k1 are reset with each destination element being updated according to the subset of write mask k1. This is useful to allow conditional re-trigger of the instruction until all the elements from a given write mask have been successfully stored.

Operation

```
// instruction works over a subset of the write mask
ktemp = SELECT_SUBSET(k1)

// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 16; n++) {
```

```
    if (ktemp[n] != 0) {
      i = 32*n
      // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[i+31:i] * SCALE)
      pointer[63:0] = $mv_t$[n]
      tmp = DownConvStore$_{f32}$(zmm1[i+31:i], SSS[2:0])
      if(DownConvStoreSizeOf$_{f32}$(SSS[2:0]) == 4) {
        MemStore(pointer) = tmp[31:0]
      } else if(DownConvStoreSizeOf$_{f32}$(SSS[2:0]) == 2) {
        MemStore(pointer) = tmp[15:0]
      } else if(DownConvStoreSizeOf$_{f32}$(SSS[2:0]) == 1) {
        MemStore(pointer) = tmp[7:0]
      }
      k1[n] = 0
    }
  }
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Memory Down-conversion: D$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float32 to float16	zmm1 {float16}	2
100	float32 to uint8	zmm1 {uint8}	1
101	float32 to sint8	zmm1 {sint8}	1
110	float32 to uint16	zmm1 {uint16}	2
111	float32 to sint16	zmm1 {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm512_i32extscatter_ps (void*, __m512i, __m512, _MM_DOWNCONV_PS_ENUM, int, int);
void _mm512_mask_i32extscatter_ps (void*, __mmask16, __m512i, __m512, _MM_DOWNCONV_PS_ENUM, int, int);
void _mm512_i32scatter_ps (void*, __m512i, __m512, int);
void _mm512_mask_i32scatter_ps (void*, __mmask16, __m512i, __m512, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form, and corresponding write-mask bit is not zero.
#GP(0)	If a memory address is in a non-canonical form, and corresponding write-mask bit is not zero. If a memory operand linear address is not aligned to element-wise data granularity dictated by the DownConv mode, and corresponding write-mask bit is not zero.
#PF(fault-code)	If a memory operand linear address produces a page fault and corresponding write-mask bit is not zero.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If using a 16 bit effective address. If ModRM.rm is different than 100b. If no write mask is provided or selected write-mask is k0.

VSCATTERPF0DPS - Scatter Prefetch Float32 Vector With Signed Dword Indices Into L1

Opcode	Instruction	Description
MVEX.512.66.0F38.W0 C6 /5 /vsib	vscatterpf0dps $U_{f32}(mv_t)$ {k1}	Scatter Prefetch float32 vector $U_{f32}(mv_t)$, using doubleword indices with T0 hint, under write-mask.

Description

Prefetches into the L1 level of cache the memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$, with scale $SCALE$, with request for ownership (exclusive). Up-conversion operand specifies the granularity used by compilers to better encode the instruction if a displacement, using disp8*N feature, is provided when specifying the address. If any memory access causes any type of memory exception, the memory access will be considered as completed (destination mask updated) and the exception ignored. Up-conversion parameter is optional, and it is used to correctly encode disp8*N.

Note the special mask behavior as only a subset of the active elements of write mask k1 are actually operated on (as denoted by function $SELECT\_SUBSET$). There are only two guarantees about the function: (a) the destination mask is a subset of the source mask (identity is included), and (b) on a given invocation of the instruction, **at least** one element (the least significant enabled mask bit) will be selected from the source mask.

Programmers should always enforce the execution of a gather/scatter instruction to be re-executed (via a loop) until the full completion of the sequence (i.e. all elements of the prefetch sequence have been prefetched and hence, the write-mask bits all are zero).

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element after up-conversion.

Note also the special mask behavior as the corresponding bits in write mask k1 are reset with each destination element being updated according to the subset of write mask k1. This is useful to allow conditional re-trigger of the instruction until all the elements from a given write mask have been successfully stored.

Note that both gather and scatter prefetches set the access bit (A) in the related TLB page entry. Scatter prefetches (which prefetch data with RFO) do not set the dirty bit (D).

Operation

```
// instruction works over a subset of the write mask
ktemp = SELECT_SUBSET(k1)

exclusive = 1
evicthintpre = MVEX.EH
```

```
// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 16; n++) {
  if (ktemp[n] != 0) {
    i = 32*n
    // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[i+31:i] * SCALE)
    pointer[63:0] = $mv_t$[n]
    FetchL1cacheLine(pointer, exclusive, evicthintpre)
    k1[n] = 0
  }
}
```

SIMD Floating-Point Exceptions

None.

Memory Up-conversion: U$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float16 to float32	[rax] {float16}	2
100	uint8 to float32	[rax] {uint8}	1
101	sint8 to float32	[rax] {sint8}	1
110	uint16 to float32	[rax] {uint16}	2
111	sint16 to float32	[rax] {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

void	_mm512_prefetch_i32extscatter_ps (void*, __m512i, _MM_UPCONV_PS_ENUM, int, int);
void	_mm512_mask_prefetch_i32extscatter_ps(void*, __mmask16, __m512i, _MM_UPCONV_PS_ENUM, int, int);
void	_mm512_prefetch_i32scatter_ps(void*, __m512i, int, int);
void	_mm512_mask_prefetch_i32scatter_ps(void*, __mmask16, __m512i, int, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If using a 16 bit effective address. If ModRM.rm is different than 100b. If no write mask is provided or selected write-mask is k0.

VSCATTERPF0HINTDPD - Scatter Prefetch Float64 Vector Hint With Signed Dword Indices

Opcode	Instruction		Description
Opcode	**Instruction**		**Description**
MVEX.512.66.0F38.W1 C6 /4 /vsib	vscatterpf0hintdpd {k1}	$U_{f64}(mv_t)$	Scatter Prefetch float64 vector $U_{f64}(mv_t)$, using doubleword indices with T0 hint, under write-mask.

Description

The instruction specifies a set of 8 float64 memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$ with scale $SCALE$ as a performance hint that a real scatter instruction with the same set of sources will be invoked. A programmer may execute this instruction before a real scatter instruction to improve its performance.

This instruction is a hint and may be speculative, and may be dropped or specify invalid addresses without causing problems or memory related faults. This instructions does not modify any kind of architectural state (including the write-mask).

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Operation

```
// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 8; n++) {
  if (k1[n] != 0) {
    i = 64*n
    j = 32*n
    // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[j+31:j] * SCALE)
    pointer[63:0] = $mv_t$[n]
    HintPointer(pointer)
  }
}
```

SIMD Floating-Point Exceptions

None.

Memory Up-conversion: U_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	8
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Intel® C/C++ Compiler Intrinsic Equivalent

None

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If using a 16 bit effective address. If ModRM.rm is different than 100b. If no write mask is provided or selected write-mask is k0.

VSCATTERPF0HINTDPS - Scatter Prefetch Float32 Vector Hint With Signed Dword Indices

Opcode	Instruction		Description
MVEX.512.66.0F38.W0 C6 /4 /vsib	vscatterpf0hintdps {k1}	$U_{f32}(mv_t)$	Scatter Prefetch float32 vector $U_{f32}(mv_t)$, using doubleword indices with T0 hint, under write-mask.

Description

The instruction specifies a set of 16 float32 memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$ with scale $SCALE$ as a performance hint that a real scatter instruction with the same set of sources will be invoked. A programmer may execute this instruction before a real scatter instruction to improve its performance.

This instruction is a hint and may be speculative, and may be dropped or specify invalid addresses without causing problems or memory related faults. This instructions does not modify any kind of architectural state (including the write-mask).

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Operation

```
// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 16; n++) {
  if (k1[n] != 0) {
    i = 32*n
    // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[i+31:i] * SCALE)
    pointer[63:0] = $mv_t$[n]
    HintPointer(pointer)
  }
}
```

SIMD Floating-Point Exceptions

None.

Memory Up-conversion: U$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax]	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float16 to float32	[rax] {float16}	2
100	uint8 to float32	[rax] {uint8}	1
101	sint8 to float32	[rax] {sint8}	1
110	uint16 to float32	[rax] {uint16}	2
111	sint16 to float32	[rax] {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

None

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1.
#UD	If processor model does not implement the specific instruction. If preceded by any REX, F0, F2, F3, or 66 prefixes. If using a 16 bit effective address. If ModRM.rm is different than 100b. If no write mask is provided or selected write-mask is k0.

VSCATTERPF1DPS - Scatter Prefetch Float32 Vector With Signed Dword Indices Into L2

Opcode	Instruction	Description
MVEX.512.66.0F38.W0 C6 /6 /vsib	vscatterpf1dps $U_{f32}(mv_t)$ {k1}	Scatter Prefetch float32 vector $U_{f32}(mv_t)$, using doubleword indices with T1 hint, under write-mask.

Description

Prefetches into the L2 level of cache the memory locations pointed by base address $BASE\_ADDR$ and doubleword index vector $VINDEX$, with scale $SCALE$, with request for ownership (exclusive). Down-conversion operand specifies the granularity used by compilers to better encode the instruction if a displacement, using disp8*N feature, is provided when specifying the address. If any memory access causes any type of memory exception, the memory access will be considered as completed (destination mask updated) and the exception ignored. Down-conversion parameter is optional, and it is used to correctly encode disp8*N.

Note the special mask behavior as only a subset of the active elements of write mask k1 are actually operated on (as denoted by function $SELECT\_SUBSET$). There are only two guarantees about the function: (a) the destination mask is a subset of the source mask (identity is included), and (b) on a given invocation of the instruction, **at least** one element (the least significant enabled mask bit) will be selected from the source mask.

Programmers should always enforce the execution of a gather/scatter instruction to be re-executed (via a loop) until the full completion of the sequence (i.e. all elements of the prefetch sequence have been prefetched and hence, the write-mask bits all are zero).

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element after down-conversion.

Note also the special mask behavior as the corresponding bits in write mask k1 are reset with each destination element being updated according to the subset of write mask k1. This is useful to allow conditional re-trigger of the instruction until all the elements from a given write mask have been successfully stored.

Note that both gather and scatter prefetches set the access bit (A) in the related TLB page entry. Scatter prefetches (which prefetch data with RFO) do not set the dirty bit (D).

Operation

```
// instruction works over a subset of the write mask
ktemp = SELECT_SUBSET(k1)

exclusive = 1
evicthintpre = MVEX.EH
```

```
// Use $mv_t$ as vector memory operand (VSIB)
for (n = 0; n < 16; n++) {
  if (ktemp[n] != 0) {
    i = 32*n
    // $mv_t$[n] = BASE_ADDR + SignExtend(VINDEX[i+31:i] * SCALE)
    pointer[63:0] = $mv_t$[n]
    FetchL2cacheLine(pointer, exclusive, evicthintpre)
    k1[n] = 0
  }
}
```

SIMD Floating-Point Exceptions

None.

Memory Down-conversion: D_{f32}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	zmm1	4
001	reserved	N/A	N/A
010	reserved	N/A	N/A
011	float32 to float16	zmm1 {float16}	2
100	float32 to uint8	zmm1 {uint8}	1
101	float32 to sint8	zmm1 {sint8}	1
110	float32 to uint16	zmm1 {uint16}	2
111	float32 to sint16	zmm1 {sint16}	2

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm512_prefetch_i32extscatter_ps (void*, __m512i, _MM_UPCONV_PS_ENUM, int, int);
void _mm512_mask_prefetch_i32extscatter_ps(void*, __mmask16, __m512i, _MM_UPCONV_PS_ENUM, int, int);
void _mm512_prefetch_i32scatter_ps(void*, __m512i, int, int);
void _mm512_mask_prefetch_i32scatter_ps(void*, __mmask16, __m512i, int, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes. If using a 16 bit effective address. If ModRM.rm is different than 100b. If no write mask is provided or selected write-mask is k0.

VSUBPD - Subtract Float64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F.W1 5C /r	vsubpd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Subtract float64 vector S_{f64}(zmm3/m_t) from float64 vector zmm2 and store the result in zmm1, under write-mask.

Description

Performs an element-by-element subtraction from float64 vector zmm2 of the float64 vector result of the swizzle/broadcast/conversion process on memory or float64 vector zmm3. The result is written into float64 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = zmm2[i+63:i] - tmpSrc3[i+63:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
(MXCSR.FZ)?  YES : NO
```

Memory Up-conversion: S$_{f64}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S$_{f64}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512d    _mm512_sub_pd (__m512d, __m512d);
__m512d    _mm512_mask_sub_pd (__m512d, __mmask8, __m512d, __m512d);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VSUBPS - Subtract Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.0F.W0 5C /r	vsubps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Subtract float32 vector S_{f32}(zmm3/m_t) from float32 vector zmm2 and store the result in zmm1, under write-mask.

Description

Performs an element-by-element subtraction from float32 vector zmm2 of the float32 vector result of the swizzle/broadcast/conversion process on memory or float32 vector zmm3. The result is written into float32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = zmm2[i+31:i] - tmpSrc3[i+31:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :

```
(MXCSR.FZ)? YES : NO
```

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_sub_ps (__m512, __m512);
__m512    _mm512_mask_sub_ps (__m512, __mmask16, __m512, __m512);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VSUBRPD - Reverse Subtract Float64 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W1 6D /r	vsubrpd zmm1 {k1}, zmm2, S_{f64}(zmm3/m_t)	Subtract float64 vector zmm2 from float64 vector S_{f64}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element subtraction of float64 vector zmm2 from the float64 vector result of the swizzle/broadcast/conversion process on memory or float64 vector zmm3. The result is written into float64 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f64}$(zmm3/$m_t$)
}

for (n = 0; n < 8; n++) {
  if(k1[n] != 0) {
    i = 64*n
    // float64 operation
    zmm1[i+63:i] = -zmm2[i+63:i] + tmpSrc3[i+63:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
(MXCSR.DAZ)?  YES : NO
```

Flush Tiny Results To Zero :

```
(MXCSR.FZ)?  YES : NO
```

Memory Up-conversion: S_{f64}

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {8to8} or [rax]	64
001	broadcast 1 element (x8)	[rax] {1to8}	8
010	broadcast 4 elements (x2)	[rax] {4to8}	32
011	reserved	N/A	N/A
100	reserved	N/A	N/A
101	reserved	N/A	N/A
110	reserved	N/A	N/A
111	reserved	N/A	N/A

Register Swizzle: S_{f64}

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 64 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

__m512d _mm512_subr_pd (__m512d, __m512d);
__m512d _mm512_mask_subr_pd (__m512d, __mmask8, __m512d, __m512d);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

VSUBRPS - Reverse Subtract Float32 Vectors

Opcode	Instruction	Description
MVEX.NDS.512.66.0F38.W0 6D /r	vsubrps zmm1 {k1}, zmm2, S_{f32}(zmm3/m_t)	Subtract float32 vector zmm2 from float32 vector S_{f32}(zmm3/m_t) and store the result in zmm1, under write-mask.

Description

Performs an element-by-element subtraction of float32 vector zmm2 from the float32 vector result of the swizzle/broadcast/conversion process on memory or float32 vector zmm3. The result is written into float32 vector zmm1.

This instruction is write-masked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
if(source is a register operand and MVEX.EH bit is 1) {
   if(SSS[2]==1) Supress_Exception_Flags() // SAE
  // SSS are bits 6-4 from the MVEX prefix encoding. For more details, see Table 2.14
   RoundingMode = SSS[1:0]
   tmpSrc3[511:0] = zmm3[511:0]
} else {
   RoundingMode = MXCSR.RC
   tmpSrc3[511:0] = SwizzUpConvLoad$_{f32}$(zmm3/$m_t$)
}

for (n = 0; n < 16; n++) {
  if(k1[n] != 0) {
    i = 32*n
    // float32 operation
    zmm1[i+31:i] = -zmm2[i+31:i] + tmpSrc3[i+31:i]
  }
}
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Denormal Handling

Treat Input Denormals As Zeros :

```
    (MXCSR.DAZ)? YES : NO
```

Flush Tiny Results To Zero :

```
    (MXCSR.FZ)? YES : NO
```

Memory Up-conversion: S$_{f32}$

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32

Register Swizzle: S$_{f32}$

MVEX.EH=0

$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element	zmm0 {aaaa}
101	broadcast b element	zmm0 {bbbb}
110	broadcast c element	zmm0 {cccc}
111	broadcast d element	zmm0 {dddd}

MVEX.EH=1

$S_2S_1S_0$	Rounding Mode Override	Usage
000	Round To Nearest (even)	, {rn}
001	Round Down (-INF)	, {rd}
010	Round Up (+INF)	, {ru}
011	Round Toward Zero	, {rz}
100	Round To Nearest (even) with SAE	, {rn-sae}
101	Round Down (-INF) with SAE	, {rd-sae}
110	Round Up (+INF) with SAE	, {ru-sae}
111	Round Toward Zero with SAE	, {rz-sae}

Intel® C/C++ Compiler Intrinsic Equivalent

```
__m512    _mm512_subr_ps (__m512,__m512);
__m512    _mm512_mask_subr_ps (__m512, __mmask16, __m512, __m512);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand linear address is not aligned to the data size granularity dictated by SwizzUpConv mode.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3]=1. If preceded by any REX, F0, F2, F3, or 66 prefixes.

Appendix A

Scalar Instruction Descriptions

In this Chapter all the special scalar instructions introduced with the Knights Corner instruction set are described.

CLEVICT0 - Evict L1 line

Opcode	Instruction	Description
VEX.128.F2.0F AE /7	clevict0 m8	Evict memory line from L1 in m8 using T0 hint.
MVEX.512.F2.0F AE /7	clevict0 m8	Evict memory line from L1 in m8 using T0 hint.

Description

Invalidates from the first-level cache the cache line containing the specified linear address (updating accordingly the cache hierarchy if the line is dirty). Note that, unlike CLFLUSH, the invalidation is not broadcasted throughout the cache coherence domain.

The MVEX form of this instruction uses disp8*64 addressing. Displacements that would normally be 8 bits according to the ModR/M byte are still 8 bits but scaled by 64 so that they have cache-line granularity. VEX forms of this instruction uses regular disp8 addressing.

This instruction is a hint intended for performance and may be speculative, thus may be dropped or specify invalid addresses without causing problems. The instruction does not produce any type of memory-related fault.

Operation

```
FlushL1CacheLine(linear_address)
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm_clevict (const void*, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

	If preceded by any REX, F0, F2, F3, or 66 prefixes. If operand is not a memory location.

CLEVICT1 - Evict L2 line

Opcode	Instruction	Description
VEX.128.F3.0F AE /7	clevict1 m8	Evict memory line from L2 in m8 using T1 hint.
MVEX.512.F3.0F AE /7	clevict1 m8	Evict memory line from L2 in m8 using T1 hint.

Description

Invalidates from the second-level cache the cache line containing the specified linear address (updating accordingly the cache hierarchy if the line is dirty). Note that, unlike CLFLUSH, the invalidation is not broadcasted throughout the cache coherence domain.

The MVEX form of this instruction uses disp8*64 addressing. Displacements that would normally be 8 bits according to the ModR/M byte are still 8 bits but scaled by 64 so that they have cache-line granularity. VEX forms of this instruction uses regular disp8 addressing.

This instruction is a hint intended for performance and may be speculative, thus may be dropped or specify invalid addresses without causing problems. The instruction does not produce any type of memory-related fault.

Operation

```
FlushL2CacheLine(linear_address)
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm_clevict (const void*, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

	If preceded by any REX, F0, F2, F3, or 66 prefixes. If operand is not a memory location.

DELAY - Stall Thread

Opcode	Instruction	Description
VEX.128.F3.0F.W0 AE /6	delay r32	Stall Thread using r32.
VEX.128.F3.0F.W1 AE /6	delay r64	Stall Thread using r64.

Description

Hints that the processor should not fetch/issue instructions for the current thread for the specified number of clock cycles in register source. The maximum number of clock cycles is limited to $2^{32} - 1$ (32 bit counter). The instructions is speculative and could be executed as a NOP by a given processor implementation.

Any of the following events will cause the processor to start fetching instructions for the delayed thread again: the counter counting down to zero, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, or the RESET# signal. The instruction may exit prematurely due to any interrupt (e.g. an interrupt on another thread on the same core).

This instruction must properly handle the case where the current clock count turns over. This can be accomplished by performing the subtraction shown below and treating the result as an unsigned number.

This instruction should prevent the issuing of additional instructions on the issuing thread as soon as possible, to avoid the otherwise likely case where another instruction on the same thread that was issued 3 or 4 clocks later has to be killed, creating a pipeline bubble.

If, on any given clock, all threads are non-runnable, then any that are non-runnable due to the execution of DELAY may or may not be treated as runnable threads.

Notes about Knights Corner implementation:

- In Knights Corner, the processor won't execute from a "delayed" thread before the delay counter has expired, even if there are non-runnable threads at any given point in time.

Operation

```
START_CLOCK = CURRENT_CLOCK_COUNT
DELAY_SLOTS = SRC
if(DELAY_SLOTS > 0xFFFFFFFF) DELAY_SLOTS = 0xFFFFFFFF
while ( (CURRENT_CLOCK_COUNT - START_CLOCK) < DELAY_SLOTS )
{
     *avoid fetching/issuing from the current thread*
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

```
void    _mm_delay_32 (unsigned int);
void    _mm_delay_64 (unsigned __int64);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

If preceded by any REX, F0, F2, F3, or 66 prefixes.
If operand is a memory location.

LZCNT - Leading Zero Count

Opcode	Instruction	Description
VEX.128.F3.0F.W0 BD /r	lzcnt r32, r32	Count the number of leading bits set to 0 in r32 (src), leaving the result in r32 (dst).
VEX.128.F3.0F.W1 BD /r	lzcnt r64, r64	Count the number of leading bits set to 0 in r64 (src), leaving the result in r64 (dst).

Description

Counts the number of leading most significant zero bits in a source operand (second operand) returning the result into a destination (first operand).

LZCNT is an extension of the BSR instruction. The key difference between LZCNT and BSR is that LZCNT provides operand size as output when source operand is zero, while in the case of BSR instruction, if source operand is zero, the content of destination operand are undefined.

ZF flag is set when the most significant set bit is bit OSIZE-1. CF is set when the source has no set bit.

Operation

```
temp = OPERAND_SIZE - 1

DEST = 0
while( (temp >= 0) AND (SRC[temp] == 0) )
{
   temp = temp - 1
   DEST = DEST + 1
}

if(DEST == OPERAND_SIZE) {
   CF = 1
} else {
   CF = 0
}

if(DEST == 0) ZF = 1
} else {
   ZF = 0
}
```

Flags Affected

- ZF flag is set to 1 in case of zero output (most significant bit of the source is set), and to 0 otherwise
- CF flag is set to 1 if input was zero and cleared otherwise.
- The PF, OF, AF and SF flags are set to 0

Intel® C/C++ Compiler Intrinsic Equivalent

unsigned int	_lzcnt_u32 (unsigned int);
__int64	_lzcnt_u64 (unsigned __int64);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

	If preceded by any REX, F0, F2, F3, or 66 prefixes. If second operand is a memory location.

POPCNT - Return the Count of Number of Bits Set to 1

Opcode	Instruction	Description
VEX.128.F3.0F.W0 B8 /r	popcnt r32, r32	Count the number of bits set to 1 in r32 (src), leaving the result in r32 (dst).
VEX.128.F3.0F.W1 B8 /r	popcnt r64, r64	Count the number of bits set to 1 in r64 (src), leaving the result in r64 (dst).

Operation

```
tmp = 0
for (i=0; i<OPERAND_SIZE; i++)
{
      if(SRC[i] == 1) tmp = tmp + 1
}
DEST = tmp
```

Flags Affected

- The ZF flag is set according to the result (if SRC==0)
- The OF, SF, AF, CF and PF flags are set to 0

Intel® C/C++ Compiler Intrinsic Equivalent

unsigned int	_mm_popcnt_u32 (unsigned int);
__int64	_mm_popcnt_u64 (unsigned __int64);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode	
	If preceded by any REX, F0, F2, F3, or 66 prefixes. If second operand is a memory location.

SPFLT - Set performance monitor filtering mask

Opcode	Instruction	Description
VEX.128.F2.0F.W0 AE /6	spflt r32	Set performance monitoring filtering mask using r32.
VEX.128.F2.0F.W1 AE /6	spflt r64	Set performance monitoring filtering mask using r64.

Description

SPFLT enables/disables performance monitoring on the currently executing thread only based on the LSB value of the source.

SPFLT instruction is a model specific instruction and is not part of Intel® Architecture. The bit(s) and register(s) modified are model-specific and may vary by processor implementation.

The PERF_SPFLT_CTRL model-specific register modified by SPFLT instruction may also be read / modified with the RDMSR / WRMSR instructions, when executing at privilege level 0.

The PERF_SPFLT_CTRL MSR is thread specific. SPFLT execution moves LSB of source (EAX) into the USR_PREF bit (bit 63) in the PERF_SPFLT_CTRL MSR. The lower N bits, called CNTR_x_SPFLT_EN (bits N-1:0, 1 per counter), in PERF_SPFLT_CTRL MSR control whether the USR_PREF bit affects enabling of performance monitoring for the corresponding counter.

SPFLT instruction does not modify the CNTR_x_SPFLT_EN bits, where as RDMSR and WRMSR read / modify all bits of the PERF_SPFLT_CTRL MSR.

Enabling Performance countering

On a per thread basis, a performance monitoring counter n is incremented if, and only if:

1. PERF_GLOBAL_CTRL[n] is set to 1
2. IA32 PerfEvtSeln[22] is set to 1 (where 'n' is the enabled counter)
3. PERF_SPFLT_CTRL[n] is set to 0, or, PERF_SPFLT_CTRL[63] (USR_PREF) is set to 1.
4. The desired event is asserted for thread id T

MSR address	Per-thread?	Name
2Fh	Y	PERF_GLOBAL_CTRL Bit 1: Enable IA32_PerfEvtSel1 Bit 0: Enable IA32_PerfEvtSel0
28h	Y	IA32_PerfEvtSel0 Bit 22: Enable counter 0
29h	Y	IA32_PerfEvtSel1 Bit 22: Enable counter 1
2Ch	Y	PERF_SPFLT_CTRL Bit 63: User Preference (USR_PREF). Bit 1: Counter 1 SPFLT Enable. Controls whether USR_PREF is used in enabling performance monitoring for counter 1 Bit 0: Counter 0 SPFLT Enable. Controls whether USR_PREF is used in enabling performance monitoring for counter 0

Operation

```
(* i is the thread ID of the current executing thread *)
PerfFilterMask[i][0] = SRC[0];
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

```
void    _mm_spflt_32 (unsigned int);
void    _mm_spflt_64 (unsigned __int64);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

#UD	If processor model does not implement SPFLT. If preceded by any REX, F0, F2, F3, or 66 prefixes. If operand is a memory location.

TZCNT - Trailing Zero Count

Opcode	Instruction	Description
VEX.128.F3.0F.W0 BC /r	tzcnt r32, r32	Count the number of trailing bits set to 0 in r32 (src), leaving the result in r32 (dst).
VEX.128.F3.0F.W1 BC /r	tzcnt r64, r64	Count the number of trailing bits set to 0 in r64 (src), leaving the result in r64 (dst).

Description

Searches the source operand (second operand) for the least significant set bit (1 bit). If a least significant 1 bit is found, its bit index is stored in the destination operand; otherwise, the destination operand is set to the operand size.

ZF flag is set when the least significant set bit is bit 0. CF is set when the source has no set bit.

Operation

```
index = 0
if( SRC[OPERAND_SIZE-1:0] == 0 )
{
      DEST = OPERAND_SIZE
      CF = 1
}
else
{
      while(SRC[index] == 0)
      {
            index = index+1
      }
      DEST = index
      CF = 0
}
```

Flags Affected

- The ZF is set according to the result
- The CF is set if SRC is zero
- The PF, OF, AF and SF flags are set to 0

Intel® C/C++ Compiler Intrinsic Equivalent

```
unsigned int    _tzcnt_u32 (unsigned int);
     __int64    _tzcnt_u64 (unsigned __int64);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

	If preceded by any REX, F0, F2, F3, or 66 prefixes. If second operand is a memory location.

TZCNTI - Initialized Trailing Zero Count

Opcode	Instruction	Description
VEX.128.F2.0F.W0 BC /r	tzcnti r32, r32	Count the number of trailing bits set to 0 between r32 (dst) and r32 (src).
VEX.128.F2.0F.W1 BC /r	tzcnti r64, r64	Count the number of trailing bits set to 0 between r64 (dst) and r64 (src).

Description

Searches the source operand (second operand) for the least significant set bit (1 bit) greater than bit DEST (where DEST is the destination operand, the first operand). If a least significant 1 bit is found, its bit index is stored in the destination operand ; otherwise, the destination operand is set to the operand size. The value of DEST is a signed offset from bit 0 of the source operand. Any negative DEST value will produce a search starting from bit 0, like TZCNT. Any DEST value equal to or greater than (OPERAND_SIZE-1) will cause the destination operand to be set to the operand size.

This instruction allows continuation of searches through bit vectors without having to mask off each least significant 1-bit before restarting, as is required with TZCNT.

The functionality of this instruction is exactly the same as for the TZCNT instruction, except that the search starts at bit DEST+1 rather than bit 0.

CF is set when the specified index goes beyond the operand size or there is no set bit between the index and the MSB bit of the source.

Operation

```
// DEST is a signed operand, no overflow
if (DEST[OSIZE-1:0] < 0) index = 0
else                     index = DEST + 1

if( ( index > OPERAND_SIZE-1 ) || ( SRC[OPERAND_SIZE-1:index] == 0 ) )
{
      DEST = OPERAND_SIZE
      CF=1
}
else
{
      while(SRC[index] == 0)
      {
            index = index+1
      }
      DEST = index
      CF=0
```

```
}
```

Flags Affected

- The ZF is set according to the result
- The CF is set if SRC is zero betwen index and MSB, or index is greater than the operand size.
- The PF, OF, AF and SF flags are set to 0

Intel® C/C++ Compiler Intrinsic Equivalent

```
    int   _mm_tzcnti_32 (int, unsigned int);
__int64   _mm_tzcnti_64 (__int64, unsigned __int64);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

	If preceded by any REX, F0, F2, F3, or 66 prefixes. If second operand is a memory location.

VPREFETCH0 - Prefetch memory line using T0 hint

Opcode	**Instruction**	**Description**
VEX.128.0F 18 /1	vprefetch0 m8	Prefetch memory line in m8 using T0 hint.
MVEX.512.0F 18 /1	vprefetch0 m8	Prefetch memory line in m8 using T0 hint.

Description

This is very similar to the existing IA-32 prefetch instruction, PREFETCH0, as described in *IA-32 Intel® Architecture Software Developer's Manual: Volume 2*. 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.

In contrast with the existing prefetch instruction, the MVEX form of this instruction uses disp8*64 addressing. Displacements that would normally be 8 bits according to the ModR/M byte are still 8 bits but scaled by 64 so that they have cache-line granularity. VEX forms of this instruction uses regular disp8 addressing.

This instruction is a hint and may be speculative, and may be dropped or specify invalid addresses without causing problems or memory related faults.

This instruction contains a set of hint attributes that modify the prefetching behavior:

exclusive: make line Exclusive in the L1 cache (unless it's already Exclusive or Modified in the L1 cache).

nthintpre (NTH): load data into the L1 nontemporal cache rather than the L1 temporal cache. Data will be loaded in the #TIDth way and made MRU. Data should still be cached normally in the L2 and higher caches.

Note that in Knights Corner, the hardware drops VPREFETCH if it hits L1 (so it becomes transparent to L2). Consequently, this instructon is not a good solution to avoid hot L1/cold L2 performance problems. Prefetches set the access bit (A) in the related TLB page entry, but prefetches with exclusive access (RFO) do not set the dirty bit (D).

PREFETCH Hint equivalence for Knights Corner hardware

Instruction	Cache Level	Non-temporal	Bring as exclusive
VPREFETCH0	L1	NO	NO
VPREFETCHNTA	L1	YES	NO
VPREFETCH1	L2	NO	NO
VPREFETCH2	L2	YES	NO
VPREFETCHE0	L1	NO	YES
VPREFETCHENTA	L1	YES	YES
VPREFETCHE1	L2	NO	YES
VPREFETCHE2	L2	YES	YES

Operation

```
exclusive = 0
nthintpre = 0
FetchL1CacheLine(effective_address, exclusive, nthintpre)
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

`void _mm_prefetch (char const*, int);`

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

	If preceded by any REX, F0, F2, F3, or 66 prefixes. If operand is not a memory location.

VPREFETCH1 - Prefetch memory line using T1 hint

Opcode	**Instruction**	**Description**
VEX.128.0F 18 /2	vprefetch1 m8	Prefetch memory line in m8 using T1 hint.
MVEX.512.0F 18 /2	vprefetch1 m8	Prefetch memory line in m8 using T1 hint.

Description

This is very similar to the existing IA-32 prefetch instruction, PREFETCH0, as described in *IA-32 Intel® Architecture Software Developer's Manual: Volume 2*. 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.

In contrast with the existing prefetch instruction, the MVEX form of this instruction uses disp8*64 addressing. Displacements that would normally be 8 bits according to the ModR/M byte are still 8 bits but scaled by 64 so that they have cache-line granularity. VEX forms of this instruction uses regular disp8 addressing.

This instruction is a hint and may be speculative, and may be dropped or specify invalid addresses without causing problems or memory related faults.

This instruction contains a set of hint attributes that modify the prefetching behavior:

exclusive: make line Exclusive in the L2 cache (unless it's already Exclusive or Modified in the L2 cache).

nthintpre (NTH): load data into the L2 nontemporal cache rather than the L2 temporal cache. Data will be loaded in the #TIDth way and made MRU. Data should still be cached normally in the L2 and higher caches.

Note that in Knights Corner, the hardware drops VPREFETCH if it hits L1 (so it becomes transparent to L2). Consequently, this instructon is not a good solution to avoid hot L1/cold L2 performance problems. Prefetches set the access bit (A) in the related TLB page entry, but prefetches with exclusive access (RFO) do not set the dirty bit (D).

PREFETCH Hint equivalence for Knights Corner hardware

Instruction	Cache Level	Non-temporal	Bring as exclusive
VPREFETCH0	L1	NO	NO
VPREFETCHNTA	L1	YES	NO
VPREFETCH1	L2	NO	NO
VPREFETCH2	L2	YES	NO
VPREFETCHE0	L1	NO	YES
VPREFETCHENTA	L1	YES	YES
VPREFETCHE1	L2	NO	YES
VPREFETCHE2	L2	YES	YES

Operation

```
exclusive = 0
nthintpre = 0
FetchL2CacheLine(effective_address, exclusive, nthintpre)
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm_prefetch (char const*, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

	If preceded by any REX, F0, F2, F3, or 66 prefixes. If operand is not a memory location.

VPREFETCH2 - Prefetch memory line using T2 hint

Opcode	**Instruction**	**Description**
VEX.128.0F 18 /3	vprefetch2 m8	Prefetch memory line in m8 using T2 hint.
MVEX.512.0F 18 /3	vprefetch2 m8	Prefetch memory line in m8 using T2 hint.

Description

This is very similar to the existing IA-32 prefetch instruction, PREFETCH0, as described in *IA-32 Intel® Architecture Software Developer's Manual: Volume 2*. 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.

In contrast with the existing prefetch instruction, the MVEX form of this instruction uses disp8*64 addressing. Displacements that would normally be 8 bits according to the ModR/M byte are still 8 bits but scaled by 64 so that they have cache-line granularity. VEX forms of this instruction uses regular disp8 addressing.

This instruction is a hint and may be speculative, and may be dropped or specify invalid addresses without causing problems or memory related faults.

This instruction contains a set of hint attributes that modify the prefetching behavior:

exclusive: make line Exclusive in the L2 cache (unless it's already Exclusive or Modified in the L2 cache).

nthintpre (NTH): load data into the L2 nontemporal cache rather than the L2 temporal cache. Data will be loaded in the #TIDth way and made MRU. Data should still be cached normally in the L2 and higher caches.

Note that in Knights Corner, the hardware drops VPREFETCH if it hits L1 (so it becomes transparent to L2). Consequently, this instructon is not a good solution to avoid hot L1/cold L2 performance problems. Prefetches set the access bit (A) in the related TLB page entry, but prefetches with exclusive access (RFO) do not set the dirty bit (D).

PREFETCH Hint equivalence for Knights Corner hardware

Instruction	Cache Level	Non-temporal	Bring as exclusive
VPREFETCH0	L1	NO	NO
VPREFETCHNTA	L1	YES	NO
VPREFETCH1	L2	NO	NO
VPREFETCH2	L2	YES	NO
VPREFETCHE0	L1	NO	YES
VPREFETCHENTA	L1	YES	YES
VPREFETCHE1	L2	NO	YES
VPREFETCHE2	L2	YES	YES

Operation

```
exclusive = 0
nthintpre = 1
FetchL2CacheLine(effective_address, exclusive, nthintpre)
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

```
void    _mm_prefetch (char const*, int);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

	If preceded by any REX, F0, F2, F3, or 66 prefixes. If operand is not a memory location.

VPREFETCHE0 - Prefetch memory line using T0 hint, with intent to write

Opcode	Instruction	Description
VEX.128.0F 18 /5	vprefetche0 m8	Prefetch memory line in m8 using T0 hint with intent to write.
MVEX.512.0F 18 /5	vprefetche0 m8	Prefetch memory line in m8 using T0 hint with intent to write.

Description

This is very similar to the existing IA-32 prefetch instruction, PREFETCH0, as described in *IA-32 Intel® Architecture Software Developer's Manual: Volume 2*. 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.

In contrast with the existing prefetch instruction, the MVEX form of this instruction uses disp8*64 addressing. Displacements that would normally be 8 bits according to the ModR/M byte are still 8 bits but scaled by 64 so that they have cache-line granularity. VEX forms of this instruction uses regular disp8 addressing.

This instruction is a hint and may be speculative, and may be dropped or specify invalid addresses without causing problems or memory related faults.

This instruction contains a set of hint attributes that modify the prefetching behavior:

exclusive: make line Exclusive in the L1 cache (unless it's already Exclusive or Modified in the L1 cache).

nthintpre (NTH): load data into the L1 nontemporal cache rather than the L1 temporal cache. Data will be loaded in the #TIDth way and made MRU. Data should still be cached normally in the L2 and higher caches.

In Knights Corner, the hardware drops VPREFETCH if it hits L1 (so it becomes transparent to L2). Consequently, this instructon is not a good solution to avoid hot L1/cold L2 performance problems. Prefetches set the access bit (A) in the related TLB page entry, but prefetches with exclusive access (RFO) do not set the dirty bit (D).

PREFETCH Hint equivalence for Knights Corner hardware

Instruction	Cache Level	Non-temporal	Bring as exclusive
VPREFETCH0	L1	NO	NO
VPREFETCHNTA	L1	YES	NO
VPREFETCH1	L2	NO	NO
VPREFETCH2	L2	YES	NO
VPREFETCHE0	L1	NO	YES
VPREFETCHENTA	L1	YES	YES
VPREFETCHE1	L2	NO	YES
VPREFETCHE2	L2	YES	YES

Operation

```
exclusive = 1
nthintpre = 0
FetchL1CacheLine(effective_address, exclusive, nthintpre)
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

```
void    _mm_prefetch (char const*, int);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

	If preceded by any REX, F0, F2, F3, or 66 prefixes. If operand is not a memory location.

VPREFETCHE1 - Prefetch memory line using T1 hint, with intent to write

Opcode	Instruction	Description
VEX.128.0F 18 /6	vprefetche1 m8	Prefetch memory line in m8 using T1 hint with intent to write.
MVEX.512.0F 18 /6	vprefetche1 m8	Prefetch memory line in m8 using T1 hint with intent to write.

Description

This is very similar to the existing IA-32 prefetch instruction, PREFETCH0, as described in *IA-32 Intel® Architecture Software Developer's Manual: Volume 2*. 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.

In contrast with the existing prefetch instruction, the MVEX form of this instruction uses disp8*64 addressing. Displacements that would normally be 8 bits according to the ModR/M byte are still 8 bits but scaled by 64 so that they have cache-line granularity. VEX forms of this instruction uses regular disp8 addressing.

This instruction is a hint and may be speculative, and may be dropped or specify invalid addresses without causing problems or memory related faults.

This instruction contains a set of hint attributes that modify the prefetching behavior:

exclusive: make line Exclusive in the L2 cache (unless it's already Exclusive or Modified in the L2 cache).

nthintpre (NTH): load data into the L2 nontemporal cache rather than the L2 temporal cache. The data will be loaded in the #TIDth way and making the data MRU. Data should still be cached normally in the L2 and higher caches.

The hardware drops VPREFETCH if it hits L1 (so it becomes transparent to L2). Consequently, this instructon is not a good solution to avoid hot L1/cold L2 performance problems. Prefetches set the access bit (A) in the related TLB page entry, but prefetches with exclusive access (RFO) do not set the dirty bit (D).

PREFETCH Hint equivalence for Knights Corner hardware

Instruction	Cache Level	Non-temporal	Bring as exclusive
VPREFETCH0	L1	NO	NO
VPREFETCHNTA	L1	YES	NO
VPREFETCH1	L2	NO	NO
VPREFETCH2	L2	YES	NO
VPREFETCHE0	L1	NO	YES
VPREFETCHENTA	L1	YES	YES
VPREFETCHE1	L2	NO	YES
VPREFETCHE2	L2	YES	YES

Operation

```
exclusive = 1
nthintpre = 0
FetchL2CacheLine(effective_address, exclusive, nthintpre)
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm_prefetch (char const*, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

	If preceded by any REX, F0, F2, F3, or 66 prefixes. If operand is not a memory location.

VPREFETCHE2 - Prefetch memory line using T2 hint, with intent to write

Opcode	**Instruction**	**Description**
VEX.128.0F 18 /7	vprefetche2 m8	Prefetch memory line in m8 using T2 hint with intent to write.
MVEX.512.0F 18 /7	vprefetche2 m8	Prefetch memory line in m8 using T2 hint with intent to write.

Description

This is very similar to the existing IA-32 prefetch instruction, PREFETCH0, as described in *IA-32 Intel® Architecture Software Developer's Manual: Volume 2*. 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.

In contrast with the existing prefetch instruction, the MVEX form of this instruction uses disp8*64 addressing. Displacements that would normally be 8 bits according to the ModR/M byte are still 8 bits but scaled by 64 so that they have cache-line granularity. VEX forms of this instruction uses regular disp8 addressing.

This instruction is a hint and may be speculative, and may be dropped or specify invalid addresses without causing problems or memory related faults.

This instruction contains a set of hint attributes that modify the prefetching behavior:

exclusive: make line Exclusive in the L2 cache (unless it's already Exclusive or Modified in the L2 cache).

nthintpre (NTH): load data into the L2 nontemporal cache rather than the L2 temporal cache. Data will be loaded in the #TIDth way and made MRU. Data should still be cached normally in the L2 and higher caches.

Note that in Knights Corner, the hardware drops VPREFETCH if it hits L1 (so it becomes transparent to L2). Consequently, this instructon is not a good solution to avoid hot L1/cold L2 performance problems. Prefetches set the access bit (A) in the related TLB page entry, but prefetches with exclusive access (RFO) do not set the dirty bit (D).

PREFETCH Hint equivalence for Knights Corner hardware

Instruction	Cache Level	Non-temporal	Bring as exclusive
VPREFETCH0	L1	NO	NO
VPREFETCHNTA	L1	YES	NO
VPREFETCH1	L2	NO	NO
VPREFETCH2	L2	YES	NO
VPREFETCHE0	L1	NO	YES
VPREFETCHENTA	L1	YES	YES
VPREFETCHE1	L2	NO	YES
VPREFETCHE2	L2	YES	YES

Operation

```
exclusive = 1
nthintpre = 1
FetchL2CacheLine(effective_address, exclusive, nthintpre)
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm_prefetch (char const*, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

If preceded by any REX, F0, F2, F3, or 66 prefixes.
If operand is not a memory location.

VPREFETCHENTA - Prefetch memory line using NTA hint, with intent to write

Opcode	Instruction	Description
VEX.128.0F 18 /4	vprefetchenta m8	Prefetch memory line in m8 using NTA hint with intent to write.
MVEX.512.0F 18 /4	vprefetchenta m8	Prefetch memory line in m8 using NTA hint with intent to write.

Description

This is very similar to the existing IA-32 prefetch instruction, PREFETCH0, as described in *IA-32 Intel® Architecture Software Developer's Manual: Volume 2*. 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.

In contrast with the existing prefetch instruction, this instruction uses disp8*64 addressing. Displacements that would normally be 8 bits according to the ModR/M byte are still 8 bits but scaled by 64 so that they have cache-line granularity.

This instruction is a hint and may be speculative, and may be dropped or specify invalid addresses without causing problems or memory related faults.

This instruction contains a set of hint attributes that modify the prefetching behavior:

exclusive: make line Exclusive in the L1 cache (unless it's already Exclusive or Modified in the L1 cache).

nthintpre (NTH): load data into the L1 nontemporal cache rather than the L1 temporal cache. The data will be loaded in the #TIDth way and making the data MRU. Data should still be cached normally in the L2 and higher caches.

The hardware drops VPREFETCH if it hits L1 (so it becomes transparent to L2). Consequently, this instructon is not a good solution to avoid hot L1/cold L2 performance problems. Prefetches set the access bit (A) in the related TLB page entry, but prefetches with exclusive access (RFO) do not set the dirty bit (D).

PREFETCH Hint equivalence for Knights Corner hardware

Instruction	Cache Level	Non-temporal	Bring as exclusive
VPREFETCH0	L1	NO	NO
VPREFETCHNTA	L1	YES	NO
VPREFETCH1	L2	NO	NO
VPREFETCH2	L2	YES	NO
VPREFETCHE0	L1	NO	YES
VPREFETCHENTA	L1	YES	YES
VPREFETCHE1	L2	NO	YES
VPREFETCHE2	L2	YES	YES

Operation

```
exclusive = 1
nthintpre = 1
FetchL1CacheLine(effective_address, exclusive, nthintpre)
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

```
void    _mm_prefetch (char const*, int);
```

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

	If preceded by any REX, F0, F2, F3, or 66 prefixes. If operand is not a memory location.

VPREFETCHNTA - Prefetch memory line using NTA hint

Opcode	**Instruction**	**Description**
VEX.128.0F 18 /0	vprefetchnta m8	Prefetch memory line in m8 using NTA hint.
MVEX.512.0F 18 /0	vprefetchnta m8	Prefetch memory line in m8 using NTA hint.

Description

This is very similar to the existing IA-32 prefetch instruction, PREFETCH0, as described in *IA-32 Intel® Architecture Software Developer's Manual: Volume 2*. 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.

In contrast with the existing prefetch instruction, the MVEX form of this instruction uses disp8*64 addressing. Displacements that would normally be 8 bits according to the ModR/M byte are still 8 bits but scaled by 64 so that they have cache-line granularity. VEX forms of this instruction uses regular disp8 addressing.

This instruction is a hint and may be speculative, and may be dropped or specify invalid addresses without causing problems or memory related faults.

This instruction contains a set of hint attributes that modify the prefetching behavior:

exclusive: make line Exclusive in the L1 cache (unless it's already Exclusive or Modified in the L1 cache).

nthintpre (NTH): load data into the L1 nontemporal cache rather than the L1 temporal cache. Data will be loaded in the #TIDth way and made MRU. Data should still be cached normally in the L2 and higher caches.

In Knights Corner, the hardware drops VPREFETCH if it hits L1 (so it becomes transparent to L2). Consequently, this instructon is not a good solution to avoid hot L1/cold L2 performance problems. Prefetches set the access bit (A) in the related TLB page entry, but prefetches with exclusive access (RFO) do not set the dirty bit (D).

PREFETCH Hint equivalence for Knights Corner hardware

Instruction	Cache Level	Non-temporal	Bring as exclusive
VPREFETCH0	L1	NO	NO
VPREFETCHNTA	L1	YES	NO
VPREFETCH1	L2	NO	NO
VPREFETCH2	L2	YES	NO
VPREFETCHE0	L1	NO	YES
VPREFETCHENTA	L1	YES	YES
VPREFETCHE1	L2	NO	YES
VPREFETCHE2	L2	YES	YES

Operation

```
exclusive = 0
nthintpre = 1
FetchL1CacheLine(effective_address, exclusive, nthintpre)
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

void _mm_prefetch (char const*, int);

Exceptions

Real-Address Mode and Virtual-8086

#UD	Instruction not available in these modes

Protected and Compatibility Mode

#UD	Instruction not available in these modes

64 bit Mode

	If preceded by any REX, F0, F2, F3, or 66 prefixes. If operand is not a memory location.

Appendix B

Knights Corner 64 bit Mode Scalar Instruction Support

In 64 bit mode, Knights Corner supports a subset of the Intel 64 Architecture instructions. The 64 bit mode instructions supported by Knights Corner are listed in this chapter.

B.1 64 bit Mode General-Purpose and X87 Instructions

Knights Corner supports most of the general-purpose register (GPR) and X87 instructions in 64 bit mode. They are listed in Table B.2.

64 bit Mode GPR and X87 Instructions in Knights Corner:

ADC	ADD	AND	BSF	BSR
BSWAP	BT	BTC	BTR	BTS
CALL	CBW	CDQ	CDQE	CLC
CLD	CLI	CLTS	CMC	CMP
CMPS	CMPSB	CMPSD	CMPSQ	CMPSW
CMPXCHG	CMPXCHG8B	CPUID	CQO	CWD
CWDE	DEC	DIV	ENTER	F2XM1
FABS	FADD	FADDP	FBLD	FBSTP
FCHS	FCLEX	FCOM	FCOMP	FCOMPP
FCOS	FDECSTP	FDIV	FDIVP	FDIVR
FDIVRP	FFREE	FIADD	FICOM	FICOMP
FIDIV	FIDIVR	FILD	FIMUL	FINCSTP
FINIT	FIST	FISTP	FISUB	FISUBR
FLD	FLD1	FLDCW	FLDENV	FLDL2E
FLDL2T	FLDLG2	FLDLN2	FLDPI	FLDZ
FMUL	FMULP	FNCLEX	FNINIT	FNOP
FNSAVE	FNSTCW	FNSTENV	FNSTSW	FPATAN
FPREM	FPREM1	FPTAN	FRNDINT	FRESTOR

FSAVE	FSCALE	FSIN	FSINCOS	FSQRT
FST	FSTCW	FSTENV	FSTP	FSTSW
FSUB	FSUBP	FSUBR	FSUBRP	FTST
FUCOM	FUCOMP	FUCOMPP	FWAIT	FXAM
FXCH	FXRSTOR	FXSAVE	FXTRACT	FYL2X
FYL2XP1	HLT	IDIV	IMUL	INC
INT	INT3	INTO	INVD	INVPLG
IRET	IRETD	JA	JAE	JB
JBE	JC	JCXZ	JE	JECXZ
JG	JGE	JL	JLE	JMP
JNA	JNAE	JNB	JNBE	JNC
JNE	JNG	JNGE	JNL	JNLE
JNO	JNP	JNS	JNZ	JO
JP	JPE	JPO	JS	JZ
LAHF	LAR	LEA	LEAVE	LFS
LGDT	LGS	LIDT	LLDT	LMSW
LOCK	LODS	LODSB	LODSD	LODSQ
LODSW	LOOP	LOOPE	LOOPNE	LOOPNZ
LOOPZ	LSL	LSS	LTR	MOV
MOV CR	MOV DR	MOVS	MOVSB	MOVSD
MOVSQ	MOVSW	MOVSX	MOVSXD	MOVZX
MUL	NEG	NOP	NOT	OR
POP	POPF	POPFQ	PUSH	PUSHF
PUSHFQ	RCL	RCR	RDMSM	RDPMC
RDTSC	REP	REPE	REPNE	REPNZ
RET	ROL	ROR	RSM	SAHF
SAL	SAR	SBB	SCAS	SCASB
SCASD	SCASQ	SCASW	SETA	SETAE
SETB	SETBE	SETC	SETE	SETG
SETGE	SETL	SETLE	SETNA	SETNAE
SETNB	SETNBE	SETNC	SETNE	SETNG
SETNGE	SETNL	SETNLE	SETNO	SETNP
SETNS	SETNZ	SETO	SETP	SETPE
SETPO	SETS	SETZ	SGDT	SHL
SHLD	SHR	SHRD	SIDT	SLDT
SMSW	STC	STD	STI	STOSB
STOSD	STOSQ	STOSW	STR	SUB
SWAPGS	SYSCALL	SYSRET	TEST	VERR
VERW	WAIT	WBINVD	WRMSR	XADD
XCHG	XLAT	XLATB	XOR	UD2

B.2 Knights Corner 64 bit Mode Limitations

In 64 bit mode, Knights Corner supports a subset of the Intel 64 Architecture instructions. The following summarizes Intel 64 Architecture instructions that are not supported in Knights Corner:

- Instructions that operate on MMX registers
- Instructions that operate on XMM registers
- Instructions that operate on YMM registers

GPR and X87 Instructions Not Supported in Knights Corner

CMOV	CMPXCHG16B	FCMOVcc	FCOMI
FCOMIP	FUCOMI	FUCOMIP	IN
INS	INSB	INSD	INSW
MONITOR	MWAIT	OUT	OUTS
OUTSB	OUTSD	OUTSW	PAUSE
SYSENTER	SYSEXIT		

B.3 LDMXCSR - Load MXCSR Register

Opcode	**Instruction**	**Description**
0F AE /2	ldmxcsr m32	Load MXCSR register from m32

Description

Loads the source operand into the MXCSR control/status register. The source operand is a 32 bit memory location. See MXCSR Control and Status Register in Chapter 10, of the IA-32 Intel Architecture Software Developers Manual, Volume 1, for a description of the MXCSR register and its contents. See chapter 3 of this document for a description of the new Knights Corner's MXCSR feature bits.

The LDMXCSR instruction is typically used in conjunction with the STMXCSR instruction, which stores the contents of the MXCSR register in memory.

The default MXCSR value at reset is 0020_0000H (DUE=1, FZ=0, RC=00, PM=0, UM=0, OM=0, ZM=0, DM=0, IM=0, DAZ=0, PE=0, UE=0, OE=0, ZE=0, DE=0, IE=0).

Any attempt to set to 1 reserved bits in control register MXCSR will produce a #GP fault:

Bit	default	Comment
MXCSR[7-12]	0	Note that this corresponds to Intel® SSE's IM/DM/ZM/OM/UM/PM
MXCSR[16-20]	0	Reserved
MXCSR[22-31]	0	Reserved

Additionally, any attempt to set MXCSR.DUE (bit 21) to 0 will produce a #GP fault:

Bit	default	Comment
MXCSR[21]	1	DUE (Disable Unmasked Exceptions) always enforced in Knights Corner

This instructions operation is the same in non-64 bit modes and 64 bit mode.

Operation

```
MXCSR = MemLoad(m32)
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

```
void    _mm_setcsr (unsigned int)
```

Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. For an attempt to set reserved bits in MXCSR
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CS.L=0 or IA32_EFER.LMA=0. If the lock prefix is used.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

B.4 FXRSTOR - Restore x87 FPU and MXCSR State

Opcode	Instruction	Description
0F AE /1	fxrstor m512byte	Restore the x87 FPU and MXCSR register state from m512byte

Description

See Intel64® Intel® Architecture Software Developer's Manual for the description of the original x86 instruction.

Reloads the x87 FPU and the MXCSR state from the 512-byte memory image specified in the source operand. This data should have been written to memory previously using the FXSAVE instruction, and in the same format as required by the operating modes. The first byte of the data should be located on a 16-byte boundary. There are three distinct layout of the FXSAVE state map: one for legacy and compatibility mode, a second format for 64 bit mode with promoted operandsize, and the third format is for 64 bit mode with default operand size.

Knights Corner follows the same layouts as described in Intel64® Intel® Architecture Software Developer's Manual.

The state image referenced with an FXRSTOR instruction must have been saved using an FXSAVE instruction or be in the same format as required by Intel64 Intel® Architecture Software Developer's Manual. Referencing a state image saved with an FSAVE, FNSAVE instruction or incompatible field layout will result in an incorrect state restoration.

The FXRSTOR instruction does not flush pending x87 FPU exceptions. To check and raise exceptions when loading x87 FPU state information with the FXRSTOR instruction, use an FWAIT instruction after the FXRSTOR instruction.

Note that XMM15-0 registers are logically aliased to the the low 128-bit portions of Knights Corner registers V15 through V0 (ZMM15-0). Therefore, FXRSTOR must restore the contents of the low 128-bit portions of registers V15 through V0.

Any attempt to set reserved bits in control register MXCSR to 1 will produce a #GP fault:

Bit	default	Comment
MXCSR[7-12]	0	Note that this corresponds to Intel® SSE's IM/DM/ZM/OM/UM/PM
MXCSR[16-19]	0	Reserved
MXCSR[20]	0	Reserved
MXCSR[22-31]	0	Reserved

Additionally, any attempt to set MXCSR.DUE (bit 21) to 0 will produce a #GP fault:

Bit	default	Comment
MXCSR[21]	1	DUE (Disable Unmasked Exceptions) always enforced in Knights Corner

Operation

```
(x87 FPU, MXCSR, XMM) = MemLoad(SRC);
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

```
void    _fxrstor64 (void*);
```

Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment. If trying to set illegal MXCSR values.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If instruction is preceded by a LOCK prefix.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

B.5 FXSAVE - Save x87 FPU and MXCSR State

Opcode	Instruction	Description
0F AE /0	fxsave m512byte	Save the x87 FPU and MXCSR register state to m512byte

Description

See Intel64® Intel® Architecture Software Developer's Manual for the description of the original x86 instruction.

Saves the current state of the x87 FPU, XMM, and MXCSR registers to a 512-byte memory location specified in the destination operand. The content layout of the 512 byte region depends on whether the processor is operating in non- 64 bit operating modes or 64 bit sub-mode of IA-32e mode.

Bytes 464:511 are available to software use. The processor does not write to bytes 464:511 of an FXSAVE area.

Knights Corner follows a similar layout as described in Intel64® Intel® Architecture Software Developer's Manual.

All bits set to 0 in the MXCSR_MASK value indicate reserved bits in the MXCSR register. Thus, if the MXCSR_MASK value is ANDd with a value to be written into the MXCSR register, the resulting value will be assured of having all its reserved bits set to 0, preventing the possibility of a general-protection exception being generated when the value is written to the MXCSR register.

Note that XMM15-0 registers are logically aliased to the the low 128-bit portions of Knights Corner registers V15 through V0 (ZMM15-0). Therefore, FXSAVE must save the contents of the low 128-bit portions of registers V15 through V0.

Operation

```
if(64 bit Mode)
{
   if(REX.W == 1)
   {
      MemStore(m512byte) = Save64BitPromotedFxsave(x87 FPU, XMM15-XMM0, MXCSR);
   }
   else {
      MemStore(m512byte) = Save64BitDefaultFxsave(x87 FPU, XMM15-XMM0, MXCSR);
   }
}
else {
   MemStore(m512byte) = SaveLegacyFxsave(x87 FPU, XMM7-XMM0, MXCSR);
```

```
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

```
void    _fxsave64 (void*);
```

Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If instruction is preceded by a LOCK prefix.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

B.6 RDPMC - Read Performance-Monitoring Counters

Opcode	Instruction	Description
0F 33	rdpmc	Read performance-monitoring counter specified by ECX into EDX:EAX.

Description

Loads the 40-bit performance-monitoring counter specified in the ECX register into registers EDX:EAX. The EDX register is loaded with the high-order 8 bits of the counter and the EAX register is loaded with the low-order 32 bits. The counter to be read is specified with an unsigned integer placed in the ECX register.

The Knights Corner co-processor has 2 performance monitoring counters per thread, specified with 0000H through 0001H, respectively, in the ECX register.

When in protected or virtual 8086 mode, the performance-monitoring counters enabled (PCE) flag in register CR4 restricts the use of the RDPMC instruction as follows. When the PCE flag is set, the RDPMC instruction can be executed at any privilege level; when the flag is clear, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDPMC instruction is always enabled.)

The performance-monitoring counters can also be read with the RDMSR instruction, when executing at privilege level 0.

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads. Appendix A, Performance-Monitoring Events, in the IA-32 Intel® Architecture Software Developers Manual, Volume 3, lists the events that can be counted for the Intel® Pentium® 4, Intel Xeon®, and earlier IA-32 processors.

The RDPMC instruction is not a serializing instruction; that is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If an exact event count is desired, software must insert a serializing instruction (such as the CPUID instruction) before and/or after the RDPMC instruction.

The RDPMC instruction can execute in 16 bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the counter, and the event count is stored in the full EAX and EDX registers.

The RDPMC instruction was introduced into the IA-32 Architecture in the Intel® Pentium® Pro processor and the Intel® Pentium® processor with Intel® MMX™ technology. The earlier Intel® Pentium® processors have performance-monitoring counters, but they must be read with the RDMSR instruction.

In 64 bit mode, RDPMC behavior is unchanged from 32 bit mode. The upper 32 bits of RAX and RDX are cleared.

Operation

```
if ( ( (ECX[31:0] >= 0) && (ECX[31:0] < 2)
   && ((CR4.PCE = 1) || (CPL = 0) || (CR0.PE = 0))
   )
{
   if(64 bit Mode)
   {
      RAX[31:0]  = PMC(ECX[31:0])[31:0]; (* 40-bit read *)
      RAX[63:32] = 0;
      RDX[31:0]  = PMC(ECX[31:0])[39:32];
      RDX[63:32] = 0;
   }
   else
   {
      EAX = PMC(ECX[31:0])[31:0]; (* 40-bit read *)
      EDX = PMC(ECX[31:0])[39:32];
   }
}
else
{
   #GP(0)
}
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

```
__int64    _rdpmc (int);
```

Exceptions

TBD

B.7 STMXCSR - Store MXCSR Register

Opcode	Instruction	Description
0F AE /3	stmxcsr m32	Store contents of MXCSR register to m32

Description

Stores the contents of the MXCSR control and status register to the destination operand. The destination operand is a 32 bit memory location.

This instructions operation is the same in non-64 bit modes and 64 bit mode.

Operation

```
MemStore(m32) = MXCSR
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

unsigned int _mm_getcsr (void)

Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CS.L=0 or IA32_EFER.LMA=0. If the lock prefix is used.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

B.8 CPUID - CPUID Identification

Opcode	Instruction	Description
0F A2	cpuid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by the input value entered in EAX.

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. The instructions output is dependent on the contents of the EAX register upon execution. For example, the following pseudo-code 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 B.4 through B.7 shows information returned, depending on the initial value loaded into the EAX register. Table B.3 shows the maximum CPUID input value recognized for each family of IA-32 processors on which CPUID is implemented. Since Intel® Pentium® 4 family of processors, two types of information are returned: *basic* and *extended* function information. Prior to that, only the basic function information was returned. The first is accessed with EAX=0000000xh while the second is accessed with EAX=8000000xh. If a value is entered for CPUID.EAX that is invalid for a particular processor, the data for the highest basic information leaf is returned.

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.

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 B.4 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 nibble of BL *)
EDX = 49656e69h (* "ineI", with i in the low nibble of DL *)
```

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 Intel® Pentium® Processors	01H	Not Implemented
Intel® Pentium® Pro and Intel® Pentium® II Processors, Intel® Celeron Processors	02H	Not Implemented
Intel® Pentium® III Processors	03H	Not Implemented
Intel® Pentium® 4 Processors	02H	80000004H
Intel® Xeon® Processors	02H	80000004H
Intel® Pentium® M Processor	02H	80000004H
Intel® Pentium® 4 Processor supporting Intel® Hyper-Threading Technology	05H	80000008H
Intel® Pentium® D Processor (8xx)	05H	80000008H
Intel® Pentium® D Processor (9xx)	06H	80000008H
Intel® Core™ Duo Processor	0AH	80000008H
Intel® Core™ 2 Duo Processor	0AH	80000008H
Intel® Xeon® Processor 3000, 3200, 5100, 5300 Series	0AH	80000008H
Knights Corner	04H	80000008H

Table B.3: Highest CPUID Source Operand for IA-32 Processors

```
ECX = 6c65746eh (* "ntel", with n in the low nibble of CL *)
```

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

When CPUID executes with EAX set to 1, version information is returned in EAX. Extended family, extended model, model, family, and processor type for the processor code-named Knights Corner is as follows:

- Extended Model: 0000B
- Extended Family: 0000_0000B
- Model: *see table*
- Family: 1011B
- Processor Type: 00B

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.

EAX	Information Provided about the Processor	Return value
0H	Basic CPUID Information	
	EAX Maximum Input Value for Basic CPUID Information	1
	EBX "Genu"	"Genu"
	ECX "ntel"	"ntel"
	EDX "ineI"	"ineI"
1H	Basic and Extended Feature Information	
	EAX Version Information: Type, Family, Model, and Stepping ID	
	Bits 3-0: Stepping Id	xxxx
	Bits 7-4: Model	0001B
	Bits 11-8: Family ID	1011B
	Bits 13-12: Type	00B
	Bits 19-16: Extended Model Id	00B
	Bits 27-20: Extended Family Id	00000000B
	EBX Bits 7-0: Brand Index	0
	Bits 15-8: CLFLUSH/CLEVICTn line size (Value x 8 = cache line size in bytes)	8
	Bits 23-16: Maximum number of logical processors in this physical package.	248
	Bits 31-24: Initial APIC ID	xxx
	ECX Extended Feature Information (see Tables B.10)	00000000H
	EDX Feature Information (see Tables B.8 and B.9)	110193FFH
2H	Cache and TLB Information	
	EAX Reserved	0
	EBX Reserved	0
	ECX Reserved	0
	EDX Reserved	0
3H	Serial Number Information	
	EAX Reserved	0
	EBX Reserved	0
	ECX Reserved	0
	EDX Reserved	0

Table B.4: Information Returned by CPUID Instruction

- CLFLUSH/CLEVICTn instruction cache line size (second byte of EBX) - this number indicates the size of the cache line flushed with CLEVICT1 instruction in 8-byte increments. This field was introduced in the Intel® 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 Intel® 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.

EAX	Information Provided about the Processor	Return value
	CPUID leaves > 3 < 80000000 are visible only when IA32_MISC_ENABLES.BOOT_NT4[bit 22] = 0 (default).	
4H	Deterministic Cache Parameters Leaf	ECX=0/1/2
	Note: 04H output also depends on the inital value in ECX.	
	EAX Bits 4-0: Cache Type (0 = Null - No more caches; 1 = Data Cache 2 = Instruction Cache, 3 = Unified Cache)	2/1/1
	Bits 7-5: Cache Level (starts at 1)	1/1/2
	Bits 8: Self Initializing cache level (does not need SW initialization)	1/1/1
	Bits 9: Fully Associative cache	0/0/0
	Bits 10: Write-Back Invalidate	0/1/1
	Bits 11: Inclusive (of lower cache levels)	0/1/1
	Bits 13-12: Reserved	0
	Bits 25-14: Maximum number of threads sharing this cache in a physical package (minus one)	*/*/*
	Bits 31-26: Maximum number of processor cores in this physical package (minus one)	*/*/*
	EBX Bits 11-00: L = System Coherency Line Size (minus 1)	63/63/63
	Bits 21-12: P = Physical Line partitions (minus 1)	0/0/0
	Bits 31-22: W = Ways of associativity (minus 1)	7/7/7
	ECX S = Number of Sets (minus 1)	63/63/1023
	EDX Reserved = 0	0

Table B.5: Information Returned by CPUID Instruction (Contd.)

- Table B.8 through Table B.9 show encodings for EDX.
- Table B.10 show encodings for ECX.

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

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

Knights Corner considers leaf 2 to be reserved, so no cache and TLB information is returned when CPUID executes with EAX set to 2.

INPUT EAX = 3: Serial Number Information

Knights Corner does not implement *Processor Serial Number* support, as signalled by feature bit CPUID.EAX[01h].EDX.PSN. Therefore, all the returned fields are considered reserved.

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).

EAX	Information Provided about the Processor	Return value
80000000H	Extended Function CPUID Information	
	EAX Maximum Input Value for Extended CPUID Information	80000008H
	EBX Reserved	0
	ECX Reserved	0
	EDX Reserved	0
80000001H	Feature Information	
	EAX Reserved	0
	EBX Reserved	0
	ECX Bit 0: LAHF/SAHF available in 64 bit mode	1
	Bits 31-1: Reserved	0
	EDX Bits 10-0: Reserved	0
	Bit 11: SYSCALL/SYSRET available (in 64 bit mode)	1
	Bits 19-12: Reserved	0
	Bit 20: Execute Disable Bit available	0
	Bits 28-21: Reserved	0
	Bit 29: Intel® 64 Technology available	1
	Bits 31-30: Reserved	0
80000002H	Processor Brand String	
	EAX Processor Brand String	0
	EBX Processor Brand String Continued	0
	ECX Processor Brand String Continued	0
	EDX Processor Brand String Continued	0
80000003H	EAX Processor Brand String Continued	0
	EBX Processor Brand String Continued	0
	ECX Processor Brand String Continued	0
	EDX Processor Brand String Continued	0
80000004H	EAX Processor Brand String Continued	0
	EBX Processor Brand String Continued	0
	ECX Processor Brand String Continued	0
	EDX Processor Brand String Continued	0
80000005H	Reserved	
	EAX Reserved	0
	EBX Reserved	0
	ECX Reserved	0
	EDX Reserved	0

Table B.6: Information Returned by CPUID Instruction. 8000000xH leafs.

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 B.5. The associated cache structures described by the different ECX descriptors are:

- ECX=0: Instruction Cache (I1)
- ECX=1: L1 Data Cache (L1)
- ECX=2: L2 Data Cache (L2)

EAX	Information Provided about the Processor		Return value
80000006H	EAX	Reserved	0
	EBX	Reserved	0
	ECX	Bits 7-0: L2 cache Line size in bytes	64
		Bits 15-12: L2 associativity field	06H
		Bits 31-16: L2 cache size in 1K units	512
	EDX	Reserved	0
80000007H	Reserved		
	EAX	Reserved	0
	EBX	Reserved	0
	ECX	Reserved	0
	EDX	Reserved	0
80000008H	Virtual/Physical Address size		
	EAX	Bits 7-0: #Physical Address Bits	40
		Bits 15-8: #Virtual Address Bits	48
	EBX	Reserved	0
	ECX	Reserved	0
	EDX	Reserved	0

Table B.7: Information Returned by CPUID Instruction. 8000000xH leafs. (Contd.)

Operation

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

 case (EAX)
 {
  EAX == 0:
    EAX = 01H;                 // Highest basic function CPUID input value
    EBX = "Genu";
    ECX = "ineI";
    EDX = "ntel";
    break;
  EAX = 2H:
    // Cache and TLB information
    EAX = 0;
    EBX = 0;
    ECX = 0;
    EDX = 0;
    break;
  EAX = 3H:
    // PSN features
    EAX = 0;
    EBX = 0;
    ECX = 0;
    EDX = 0;
    break;
  EAX = 4H:
```

```
    // Deterministic Cache Parameters Leaf;
    EAX = *see table*
    EBX = *see table*
    ECX = *see table*
    EDX = *see table*
    break;
  EAX = 20000000H;
    EAX = 01H;                 // Reserved
    EBX = 0;                   // Reserved
    ECX = 0;                   // Reserved
    EDX = 0;                   // Reserved
    break;
  EAX = 20000001H;
    EAX = 0;                   // Reserved
    EBX = 0;                   // Reserved
    ECX = 0;                   // Reserved
    EDX = 00000010H;           // Reserved
    break;
  EAX = 80000000H;
    // Extended leaf
    EAX = 08H;                 // Highest extended function CPUID input value
    EBX = 0;                   // Reserved
    ECX = 0;                   // Reserved
    EDX = 0;                   // Reserved
    break;
  EAX = 80000001H;
    EAX = 0;                   // Reserved
    EBX = 0;                   // Reserved
    ECX[0]      = 1;           // LAHF/SAHF support in 64 bit mode
    ECX[31:1]   = 0;           // Reserved
    EDX[10:0]   = 0;           // Reserved
    EDX[11]     = 1;           // SYSCALL/SYSRET available in 64 bit mode
    EDX[19:12] = 0;            // Reserved
    EDX[20]     = 0;           // Execute Disable Bit available
    EDX[28:21] = 0;            // Reserved
    EDX[29]     = 1;           // Intel(R) 64 Technology available
    EDX[31:30] = 0;            // Reserved
    break;
  EAX = 80000002H;
    EAX = 0;                   // Processor Brand String
    EBX = 0;                   // Processor Brand String Continued
    ECX = 0;                   // Processor Brand String Continued
    EDX = 0;                   // Processor Brand String Continued
    break;
  EAX = 80000003H;
    EAX = 0;                   // Processor Brand String Continued
    EBX = 0;                   // Processor Brand String Continued
    ECX = 0;                   // Processor Brand String Continued
    EDX = 0;                   // Processor Brand String Continued
    break;
  EAX = 80000004H;
    EAX = 0;                   // Processor Brand String Continued
```

```
      EBX = 0;                   // Processor Brand String Continued
      ECX = 0;                   // Processor Brand String Continued
      EDX = 0;                   // Processor Brand String Continued
      break;
    EAX = 80000005H;
      EAX = 0;                   // Reserved
      EBX = 0;                   // Reserved
      ECX = 0;                   // Reserved
      EDX = 0;                   // Reserved
      break;
    EAX = 80000006H;
      EAX = 0;                   // Reserved
      EBX = 0;                   // Reserved
      ECX[7:0]   = 64;           // L2 cache Line size in bytes
      ECX[15:12] = 6;            // L2 associativity field (8-way)
      ECX[31:16] = 256;          // L2 cache size in 1K units
      EDX = 0;                   // Reserved
      break;
    EAX = 80000007H;
      EAX = 0;                   // Reserved
      EBX = 0;                   // Reserved
      ECX = 0;                   // Reserved
      EDX = 0;                   // Reserved
      break;
    EAX = 80000008H;
      EAX[7:0]   = 40;           // Physical Address bits
      EAX[15:8]  = 48;           // Virtual Address bits
      EAX[31:16] = 0;            // Reserved
      EBX = 0;                   // Reserved
      ECX = 0;                   // Reserved
      EDX = 0;                   // Reserved
      break;
    default, EAX == 1H:
      EAX[3:0]   = Stepping ID;
      EAX[7:4]   = *see table*   // Model
      EAX[11:8]  = 1011B;        // Family
      EAX[13:12] = 00B;          // Processor type
      EAX[15:14] = 00B;          // Reserved
      EAX[19:16] = 0000B;        // Extended Model
      EAX[23:20] = 00000000B;    // Extended Family
      EAX[31:24] = 00H;          // Reserved;
      EBX[7:0]   = 00H;          // Brand Index (* Reserved if the value is zero *)
      EBX[15:8]  = 8;            // CLEVICT1/CLFLISH Line Size (x8)
      EBX[23:16] = 248;          // Maximum number of logical processors
      EBX[31:24] = Initial Apic ID;
      ECX = 00000000H;           // Feature flags
      EDX = 110193FFH;           // Feature flags
      break;
  }
```

Flags Affected

None.

Intel® C/C++ Compiler Intrinsic Equivalent

None

Exceptions

None.

Bit #	Mnemonic	Description	Return Value
0	FPU	**Floating-point Unit On-Chip.** The processor contains an x87 FPU.	1
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.	1
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.	1
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.	1
4	TSC	**Time Stamp Counter.** The RDTSC instruction is supported, including CR4.TSD for controlling privilege.	1
5	MSR	**Model Specific Registers RDMSR and WRMSR Instructions.** The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.	1
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.	1
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.	1
8	CX8	**CMPXCHG8B Instruction.** The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).	1
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	0
11	SEP	**SYSENTER and SYSEXIT Instructions.** The SYSENTER and SYSEXIT and associated MSRs are supported.	0
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.	1
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.	0
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.	0

Table B.8: Feature Information Returned in the EDX Register (CPUID.EAX[01h].EDX)

Bit #	Mnemonic	Description	Return Value
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	0
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.	1
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.	0
18	PSN	**Processor Serial Number.** The processor supports the 96-bit processor identification number feature and the feature is enabled.	0
19	CLFSH	**CLFLUSH Instruction**. CLFLUSH Instruction is supported.	0
20	Reserved	Reserved	0
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 15, Debugging and Performance Monitoring, in the IA-32 Intel® Architecture Software Developers Manual, Volume 3).	0
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.	0
23	Intel® MMX™	**Intel® MMX™ Technology**. The processor supports the Intel® MMX™ technology.	0
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.	1
25	Intel® SSE	**Intel® SSE**. The processor supports the Intel® SSE extensions.	0
26	Intel® SSE2	**Intel® SSE2**. The processor supports the Intel® SSE2 extensions.	0
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.	0
28	HTT	**Multi-Threading**. The physical processor package is capable of supporting more than one logical processor.	1
29	TM	**Thermal Monitor**. The processor implements the thermal monitor automatic thermal control circuitry (TCC).	0
30	Reserved	Reserved	0
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.	0

Table B.9: Feature Information Returned in the EDX Register (CPUID.EAX[01h].EDX) (Contd.)

Bit #	Mnemonic	Description	Return Value
0	Intel® SSE3	Streaming SIMD Extensions 3 (SSE3). A value of 1 indicates the processor supports this technology.	0
1-2	Reserved	Reserved	0
3	MONITOR	MONITOR/MWAIT. A value of 1 indicates the processor supports this feature.	0
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.	0
5	VMX	Virtual Machine Extensions. A value of 1 indicates that the processor supports this technology.	0
6	Reserved	Reserved	0
7	EST	Enhanced Intel® SpeedStep® technology. A value of 1 indicates that the processor supports this technology.	0
8	TM2	Thermal Monitor 2. A value of 1 indicates whether the processor supports this technology.	0
9	SSSE3	Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 1 indicates the processor supports this technology.	0
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.	0
11-12	Reserved	Reserved	0
13	CMPXCHG16B	CMPXCHG16B Available. A value of 1 indicates that the feature is available. See the CMPXCHG8B/CMPXCHG16BCompare and Exchange Bytes section in Volume 2A.	0
14	xTPR Update Control	xTPR Update Control. A value of 1 indicates that the processor supports changing IA32_MISC_ENABLES[bit 23].	0
15	PDCM	Perf/Debug Capability MSR. A value of 1 indicates that the processor supports the performance and debug feature indication MSR	0
18 - 16	Reserved	Reserved	0
19	Intel® SSE4.1	Intel® Streaming SIMD Extensions 4.1 (Intel® SSE4.1). A value of 1 indicates the processor supports this technology.	0
20	Intel® SSE4.2	Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2). A value of 1 indicates the processor supports this technology.	0
22 - 21	Reserved	Reserved	0
23	POPCNT	POPCNT. A value of 1 indicates the processor supports the POPCNT instruction.	0 [a]
31 - 24	Reserved	Reserved	0

Table B.10: Feature Information Returned in the ECX Register (CPUID.EAX[01h].ECX)

---

[a] CPUID bit 23 erroneously indicates that POPCNT is not supported. Knights Corner does support the POPCNT instruction. See Appendix A for more information.

Appendix C

Floating-Point Exception Summary

C.1 Instruction floating-point exception summary

Table C.3 shows all those instruction that can generate a floating-point exception. Each type of exception is shown per instruction. For each table entry you will find one of the following symbols:

- Nothing : Exception of that type cannot be produced by that instruction.
- Y_{both}: The instruction can produce that exception. The exception may be produced by either the operation or the data-type conversion applied to memory operand.
- Y_{conv}: The instruction can produce that exception. That exception can only be produced by the data-type conversion applied to memory operand.
- Y_{oper}: The instruction can produce that exception. The exception can only be produced by the operation. The data-type conversion applied to the memory operand cannot produce any exception.

Instruction	#I	#D	#Z	#O	#U	#P
vaddpd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vaddps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vaddnpd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vaddnps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vaddsetsps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vblendmps	Y_{conv}					
vbroadcastf32x4	Y_{conv}					
vbroadcastss	Y_{conv}					
vcmppd	Y_{both}	Y_{oper}				
vcmpps	Y_{both}	Y_{oper}				
vcvtpd2ps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vcvtps2pd	Y_{both}	Y_{oper}				
vcvtfxpntdq2ps						Y_{oper}
vcvtfxpntpd2dq	Y_{both}					Y_{oper}
vcvtfxpntpd2udq	Y_{both}					Y_{oper}

Instruction	#I	#D	#Z	#O	#U	#P
vcvtfxpntps2dq	Y_{both}					Y_{oper}
vcvtfxpntps2udq	Y_{both}					Y_{oper}
vcvtfxpntudq2ps						Y_{oper}
vexp223ps				Y_{oper}		
vfixupnanpd	Y_{both}					
vfixupnanps	Y_{both}					
vfmadd132pd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfmadd132ps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfmadd213pd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfmadd213ps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfmadd231pd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfmadd231ps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfmadd233ps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfmsub132pd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfmsub132ps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfmsub213pd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfmsub213ps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfmsub231pd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfmsub231ps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfnmadd132pd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfnmadd132ps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfnmadd213pd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfnmadd213ps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfnmadd231pd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfnmadd231ps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfnmsub132pd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfnmsub132ps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfnmsub213pd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfnmsub213ps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfnmsub231pd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vfnmsub231ps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vgatherdps	Y_{conv}					
vgetexppd	Y_{both}	Y_{oper}				
vgetexpps	Y_{both}	Y_{oper}				
vgetmantpd	Y_{both}	Y_{oper}				
vgetmantps	Y_{both}	Y_{oper}				
vgmaxpd	Y_{both}	Y_{oper}				
vgmaxps	Y_{both}	Y_{oper}				
vgmaxabsps	Y_{both}	Y_{oper}				
vgminpd	Y_{both}	Y_{oper}				
vgminps	Y_{both}	Y_{oper}				
vloadunpackhps	Y_{conv}					
vloadunpacklps	Y_{conv}					
vlog2ps	Y_{both}		Y_{oper}			
vmovaps (load)	Y_{conv}					
vmovaps (store)	Y_{conv}	Y_{conv}		Y_{conv}	Y_{conv}	Y_{conv}
vmulpd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vmulps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}

Instruction	#I	#D	#Z	#O	#U	#P
vpackstorehps	Y_{conv}	Y_{conv}		Y_{conv}	Y_{conv}	Y_{conv}
vpackstorelps	Y_{conv}	Y_{conv}		Y_{conv}	Y_{conv}	Y_{conv}
vrcp23ps	Y_{both}		Y_{oper}			
vrndfxpntpd	Y_{both}					Y_{oper}
vrndfxpntps	Y_{both}					Y_{oper}
vrsqrt23ps	Y_{both}		Y_{oper}			
vscaleps	Y_{oper}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vscatterdps	Y_{conv}	Y_{conv}		Y_{conv}	Y_{conv}	Y_{conv}
vsubpd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vsubps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vsubrpd	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}
vsubrps	Y_{both}	Y_{oper}		Y_{oper}	Y_{oper}	Y_{oper}

C.2 Conversion floating-point exception summary

Float-to-float		
Float16 to float32	SwizzUpConv/UpConv	Invalid (on SNaN)
Float32 to float64	VCVTPS2PD	Invalid (on SNaN), Denormal
Float32 to float16	DownConv	Invalid (on SNaN), Overflow, Underflow, Precision, Denormal
Float64 to float32	VCVTPD2PS	Invalid (on SNaN), Overflow, Underflow, Precision, Denormal
Integer-to-float		
Uint8/16 to float32	UpConv	None
Sint8/16 to float32	UpConv	None
Uint32 to float32	VCVTFXPNTUDQ2PS	Precision
Sint32 to float32	VCVTFXPNTDQ2PS	Precision
Uint32 to float64	VCVTUDQ2PD	None
Sint32 to float64	VCVTDQ2PD	None
Float-to-integer		
Float32 to uint8/16	DownConv	Invalid (on NaN, out-of-range), Precision (if in-range but input not integer)
Float32 to sint8/16	DownConv	Invalid (on NaN, out-of-range), Precision (if in-range but input not integer)
Float32 to uint32	VCVTFXPNTPS2UDQ	Invalid (on NaN, out-of-range), Precision (if in-range but input not integer)
Float32 to sint32	VCVTFXPNTPS2DQ	Invalid (on NaN, out-of-range), Precision (if in-range but input not integer)
Float64 to uint32	VCVTFXPNTPD2UDQ	Invalid (on NaN, out-of-range), Precision (if in-range but input not integer)
Float64 to sint32	VCVTFXPNTPD2DQ	Invalid (on NaN, out-of-range), Precision (if in-range but input not integer)

Out-of-range values are dependent on operation definition and rounding mode. Table C.3 and Table C.4 describe maximum and minimum allowed values for float to integer and float to float conversion respectively. Please note that presented ranges are considered after ``Denormals Are Zero (DAZ)'' are applied.

Those entries in Table C.4 labelled with an asterisk(*), are not required for Knights Corner.

C.3 Denormal behavior

Instruction	Treat Input Denormals As Zeros	Flush Tiny Results To Zero
vaddpd	MXCSR.DAZ	MXCSR.FZ
vaddps	MXCSR.DAZ	MXCSR.FZ
vaddnpd	MXCSR.DAZ	MXCSR.FZ
vaddnps	MXCSR.DAZ	MXCSR.FZ
vaddsetsps	MXCSR.DAZ	MXCSR.FZ
vblendmpd	NO	NO
vblendmps	NO	NO
vcmppd	MXCSR.DAZ	Not Applicable
vcmpps	MXCSR.DAZ	Not Applicable
vcvtdq2pd	Not Applicable	Not Applicable
vcvtpd2ps	MXCSR.DAZ	MXCSR.FZ
vcvtps2pd	MXCSR.DAZ	Not Applicable
vcvtudq2pd	Not Applicable	Not Applicable
vcvtfxpntdq2ps	Not Applicable	Not Applicable
vcvtfxpntpd2dq	MXCSR.DAZ	Not Applicable
vcvtfxpntpd2udq	MXCSR.DAZ	Not Applicable
vcvtfxpntps2dq	MXCSR.DAZ	Not Applicable
vcvtfxpntps2udq	MXCSR.DAZ	Not Applicable
vcvtfxpntudq2ps	Not Applicable	Not Applicable
vexp223ps	Not Applicable	YES
vfixupnanpd	MXCSR.DAZ	NO
vfixupnanps	MXCSR.DAZ	NO
vfmadd132pd	MXCSR.DAZ	MXCSR.FZ
vfmadd132ps	MXCSR.DAZ	MXCSR.FZ
vfmadd213pd	MXCSR.DAZ	MXCSR.FZ
vfmadd213ps	MXCSR.DAZ	MXCSR.FZ
vfmadd231pd	MXCSR.DAZ	MXCSR.FZ
vfmadd231ps	MXCSR.DAZ	MXCSR.FZ
vfmadd233ps	MXCSR.DAZ	MXCSR.FZ
vfmsub132pd	MXCSR.DAZ	MXCSR.FZ
vfmsub132ps	MXCSR.DAZ	MXCSR.FZ
vfmsub213pd	MXCSR.DAZ	MXCSR.FZ
vfmsub213ps	MXCSR.DAZ	MXCSR.FZ
vfmsub231pd	MXCSR.DAZ	MXCSR.FZ
vfmsub231ps	MXCSR.DAZ	MXCSR.FZ
vfnmadd132pd	MXCSR.DAZ	MXCSR.FZ

Instruction	Treat Input Denormals As Zeros	Flush Tiny Results To Zero
vfnmadd132ps	MXCSR.DAZ	MXCSR.FZ
vfnmadd213pd	MXCSR.DAZ	MXCSR.FZ
vfnmadd213ps	MXCSR.DAZ	MXCSR.FZ
vfnmadd231pd	MXCSR.DAZ	MXCSR.FZ
vfnmadd231ps	MXCSR.DAZ	MXCSR.FZ
vfnmsub132pd	MXCSR.DAZ	MXCSR.FZ
vfnmsub132ps	MXCSR.DAZ	MXCSR.FZ
vfnmsub213pd	MXCSR.DAZ	MXCSR.FZ
vfnmsub213ps	MXCSR.DAZ	MXCSR.FZ
vfnmsub231pd	MXCSR.DAZ	MXCSR.FZ
vfnmsub231ps	MXCSR.DAZ	MXCSR.FZ
vgatherdpd	NO	NO
vgatherdps	NO	NO
vgatherpf0dps	NO	NO
vgatherpf0hintdpd	NO	NO
vgatherpf0hintdps	NO	NO
vgatherpf1dps	NO	NO
vgetexppd	MXCSR.DAZ	Not Applicable
vgetexpps	MXCSR.DAZ	Not Applicable
vgetmantpd	MXCSR.DAZ	Not Applicable
vgetmantps	MXCSR.DAZ	Not Applicable
vgmaxpd	MXCSR.DAZ	NO
vgmaxps	MXCSR.DAZ	NO
vgmaxabsps	MXCSR.DAZ	NO
vgminpd	MXCSR.DAZ	NO
vgminps	MXCSR.DAZ	NO
vloadunpackhpd	NO	NO
vloadunpackhps	NO	NO
vloadunpacklpd	NO	NO
vloadunpacklps	NO	NO
vlog2ps	YES	YES
vmovapd (load)	NO	NO
vmovapd (store)	NO (DAZ*)	NO
vmovaps (load)	NO	NO
vmovaps (store)	NO (DAZ*)	NO
vmovnrapd (load)	NO	NO
vmovnrapd (store)	NO (DAZ*)	NO
vmovnraps (load)	NO	NO
vmovnraps (store)	NO (DAZ*)	NO
vmovnrngoapd (load)	NO	NO
vmovnrngoapd (store)	NO (DAZ*)	NO
vmovnrngoaps (load)	NO	NO
vmovnrngoaps (store)	NO (DAZ*)	NO
vmulpd	MXCSR.DAZ	MXCSR.FZ
vmulps	MXCSR.DAZ	MXCSR.FZ
vpackstorehpd	NO (DAZ*)	NO
vpackstorehps	NO (DAZ*)	NO
vpackstorelpd	NO (DAZ*)	NO

Instruction	Treat Input Denormals As Zeros	Flush Tiny Results To Zero
vpackstorelps	NO (DAZ*)	NO
vrcp23ps	YES	YES
vrndfxpntpd	MXCSR.DAZ	NO
vrndfxpntps	MXCSR.DAZ	NO
vrsqrt23ps	YES	YES
vscaleps	MXCSR.DAZ	MXCSR.FZ
vscatterdpd	NO (DAZ*)	NO
vscatterdps	NO (DAZ*)	NO
vscatterpf0dps	NO	NO
vscatterpf0hintdpd	NO	NO
vscatterpf0hintdps	NO	NO
vscatterpf1dps	NO	NO
vsubpd	MXCSR.DAZ	MXCSR.FZ
vsubps	MXCSR.DAZ	MXCSR.FZ
vsubrpd	MXCSR.DAZ	MXCSR.FZ
vsubrps	MXCSR.DAZ	MXCSR.FZ

(*) FP32 down-conversion obeys MXCSR.DAZ

Conversion	Context	Rounding	Max	Min
Float32 to uint8	DownConv	RN	0x437f7fff (255.5 - 1ulp)	0xbf000000 (-0.5)
Float32 to sint8	DownConv	RN	0x42feffff (127.5 - 1ulp)	0xc3008000 (-128.5)
Float32 to uint16	DownConv	RN	0x477fff7f (65535.5 - 1ulp)	0xbf000000 (-0.5)
Float32 to sint16	DownConv	RN	0x46fffeff (32767.5 - 1ulp)	0xc7000080 (-32768.5)
Float32 to uint32	VCVTFXPNTPS2UDQ	RN	0x4f7fffff (2^32 - 1ulp)	0xbf000000 (-0.5)
Float32 to uint32	VCVTFXPNTPS2UDQ	RD	0x4f7fffff (2^32 - 1ulp)	0x80000000 (-0.0)
Float32 to uint32	VCVTFXPNTPS2UDQ	RU	0x4f7fffff (2^32 - 1ulp)	0xbf7fffff (-1.0 + 1ulp)
Float32 to uint32	VCVTFXPNTPS2UDQ	RZ	0x4f7fffff (2^32 - 1ulp)	0xbf7fffff (-1.0 + 1ulp)
Float32 to sint32	VCVTFXPNTPS2DQ	RN	0x4effffff (2^31 - 1ulp)	0xcf000000 (-2^31)
Float32 to sint32	VCVTFXPNTPS2DQ	RD	0x4effffff (2^31 - 1ulp)	0xcf000000 (-2^31)
Float32 to sint32	VCVTFXPNTPS2DQ	RU	0x4effffff (2^31 - 1ulp)	0xcf000000 (-2^31)
Float32 to sint32	VCVTFXPNTPS2DQ	RZ	0x4effffff (2^31 - 1ulp)	0xcf000000 (-2^31)
Float64 to uint32	VCVTFXPNTPD2UDQ	RN	0x41efffffffefffff (2^32 - 0.5 - 1ulp)	0xbfe0000000000000 (-0.5)
Float64 to uint32	VCVTFXPNTPD2UDQ	RD	0x41efffffffffffff (2^32 - 1ulp)	0x8000000000000000 (-0.0)
Float64 to uint32	VCVTFXPNTPD2UDQ	RU	0x41efffffffe00000 (2^32 - 1.0)	0xbfefffffffffffff (-1.0 + 1ulp)
Float64 to uint32	VCVTFXPNTPD2UDQ	RZ	0x41efffffffffffff (2^32 - 1ulp)	0xbfefffffffffffff (-1.0 + 1ulp)
Float64 to sint32	VCVTFXPNTPD2DQ	RN	0x41dfffffffdfffff (2^31 - 0.5 - 1ulp)	0xc1e0000000100000 (-2^31 - 0.5)
Float64 to sint32	VCVTFXPNTPD2DQ	RD	0x41dfffffffffffff (2^31 - 1ulp)	0xc1e0000000000000 (-2^31)
Float64 to sint32	VCVTFXPNTPD2DQ	RU	0x41dfffffffc00000 (2^31 - 1.0)	0xc1e00000001fffff (-2^31 - 1.0 + 1ulp)
Float64 to sint32	VCVTFXPNTPD2DQ	RZ	0x41dfffffffffffff (2^31 - 1ulp)	0xc1e00000001fffff (-2^31 - 1.0 + 1ulp)

Table C.3: Float-to-integer Max/Min Valid Range

Case	Rounding	Max pos arg w/o overflow	Min pos arg w/ overflow
Float32 to float16	RN	0x477fefff (65520.0 - 1ulp)	0x477ff000 (65520.0)
	RD*	0x477fffff (65536.0 - 1ulp)	0x47800000 (65536.0)
	RU*	0x477fe000 (65504.0)	0x477fe001 (65504.0 + 1ulp)
	RZ	0x477fffff (65536.0 - 1ulp)	0x47800000 (65536.0)
Float64 to float32	RN	0x47efffffefffffff ($2^{128} - 2^{103} - 1$ulp)	0x47effffff0000000 ($2^{128} - 2^{103}$)
	RD	0x47efffffffffffff ($2^{128} - 1$ulp)	0x47f0000000000000 ($2^{128.0}$)
	RU	0x47efffffe0000000 ($2^{128} - 2^{104}$)	0x47efffffe0000001 ($2^{128} - 2^{104} + 1$ulp)
	RZ	0x47efffffffffffff ($2^{128} - 1$ulp)	0x47f0000000000000 ($2^{128.0}$)

Case	Rounding	Max neg arg w/o overflow	Min neg arg w/ overflow
Float32 to float16	RN	0xc77fefff (-65520.0 + 1ulp)	0xc77ff000 (-65520.0)
	RD*	0xc77fe000 (-65504.0)	0xc77fe001 (-65504.0 - 1ulp)
	RU*	0xc77fffff (-65536.0 + 1ulp)	0xc7800000 (-65536.0)
	RZ	0xc77fffff (-65536.0 + 1ulp)	0xc7800000 (-65536.0)
Float64 to float32	RN	0xc7efffffefffffff ($-2^{128} + 2^{103} + 1$ulp)	0xc7effffff0000000 ($-2^{128} + 2^{103}$)
	RD	0xc7efffffe0000000 ($-2^{128} + 2^{104}$)	0xc7efffffe0000001 ($-2^{128} + 2^{104} - 1$ulp)
	RU	0xc7efffffffffffff ($-2^{128} + 1$ulp)	0xc7f0000000000000 ($-2^{128.0}$)
	RZ	0xc7efffffffffffff ($-2^{128} + 1$ulp)	0xc7f0000000000000 ($-2^{128.0}$)

Table C.4: Float-to-float Max/Min Valid Range

Appendix D

Instruction Attributes and Categories

In this Appendix we enumerate instruction attributes and categories

D.1 Conversion Instruction Families

D.1.1 D_{f32} Family of Instructions

VMOVAPS
VMOVNRAPS
VMOVNRNGOAPS
VPACKSTOREHPS
VPACKSTORELPS
VSCATTERDPS
VSCATTERPF1DPS

D.1.2 D_{f64} Family of Instructions

VMOVAPD
VMOVNRAPD
VMOVNRNGOAPD
VPACKSTOREHPD
VPACKSTORELPD
VSCATTERDPD

D.1.3 D_{i32} Family of Instructions

VMOVDQA32
VPACKSTOREHD
VPACKSTORELD
VPSCATTERDD

D.1.4 D_{i64} Family of Instructions

VMOVDQA64
VPACKSTOREHQ
VPACKSTORELQ
VPSCATTERDQ

D.1.5 S_{f32} Family of Instructions

VADDNPS
VADDPS
VADDSETSPS
VBLENDMPS
VCMPPS
VCVTFXPNTPS2DQ
VCVTFXPNTPS2UDQ
VCVTPS2PD
VFMADD132PS
VFMADD213PS
VFMADD231PS
VFMADD233PS
VFMSUB132PS
VFMSUB213PS
VFMSUB231PS
VFNMADD132PS
VFNMADD213PS
VFNMADD231PS
VFNMSUB132PS
VFNMSUB213PS
VFNMSUB231PS
VGETEXPPS
VGETMANTPS
VGMAXABSPS
VGMAXPS
VGMINPS
VMULPS
VRNDFXPNTPS
VSUBPS
VSUBRPS

D.1.6 S_{f64} Family of Instructions

VADDNPD
VADDPD
VBLENDMPD
VCMPPD
VCVTFXPNTPD2DQ
VCVTFXPNTPD2UDQ
VCVTPD2PS
VFMADD132PD
VFMADD213PD
VFMADD231PD
VFMSUB132PD
VFMSUB213PD
VFMSUB231PD
VFNMADD132PD
VFNMADD213PD
VFNMADD231PD
VFNMSUB132PD
VFNMSUB213PD
VFNMSUB231PD
VGETEXPPD
VGETMANTPD
VGMAXPD
VGMINPD
VMULPD
VRNDFXPNTPD
VSUBPD
VSUBRPD

D.1.7 S_{i32} Family of Instructions

VCVTDQ2PD
VCVTFXPNTDQ2PS
VCVTFXPNTUDQ2PS
VCVTUDQ2PD
VFIXUPNANPS
VPADCD
VPADDD
VPADDSETCD
VPADDSETSD
VPANDD
VPANDND
VPBLENDMD
VPCMPD
VPCMPEQD
VPCMPGTD
VPCMPLTD
VPCMPUD
VPMADD231D
VPMADD233D
VPMAXSD
VPMAXUD
VPMINSD
VPMINUD
VPMULHD
VPMULHUD
VPMULLD
VPORD
VPSBBD
VPSBBRD
VPSLLD
VPSLLVD
VPSRAD
VPSRAVD
VPSRLD
VPSRLVD
VPSUBD
VPSUBRD
VPSUBRSETBD
VPSUBSETBD
VPTESTMD
VPXORD
VSCALEPS

D.1.8 S_{i64} Family of Instructions

VFIXUPNANPD
VPANDNQ
VPANDQ
VPBLENDMQ
VPORQ
VPXORQ

D.1.9 U_{f32} Family of Instructions

VBROADCASTF32X4
VBROADCASTSS
VGATHERDPS
VGATHERPF0DPS
VGATHERPF0HINTDPS
VGATHERPF1DPS
VLOADUNPACKHPS
VLOADUNPACKLPS
VMOVAPS
VMOVNRAPS
VMOVNRNGOAPS
VSCATTERPF0DPS
VSCATTERPF0HINTDPS

D.1.10 U_{f64} Family of Instructions

VBROADCASTF64X4
VBROADCASTSD
VGATHERDPD
VGATHERPF0HINTDPD
VLOADUNPACKHPD
VLOADUNPACKLPD
VMOVAPD
VMOVNRAPD
VMOVNRNGOAPD
VSCATTERPF0HINTDPD

D.1.11 U_{i32} Family of Instructions

VBROADCASTI32X4
VLOADUNPACKHD
VLOADUNPACKLD
VMOVDQA32
VPBROADCASTD
VPGATHERDD

D.1.12 U_{i64} Family of Instructions

VBROADCASTI64X4
VLOADUNPACKHQ
VLOADUNPACKLQ
VMOVDQA64
VPBROADCASTQ
VPGATHERDQ

Appendix E

Non-faulting Undefined Opcodes

The following opcodes are non-faulting and have undefined behavior:

- MVEX.512.0F38.W0 D2 /r
- MVEX.512.0F38.W0 D3 /r
- MVEX.512.0F38.W0 D6 /r
- MVEX.512.0F38.W0 D7 /r
- MVEX.512.66.0F38.W0 48 /r
- MVEX.512.66.0F38.W0 49 /r
- MVEX.512.66.0F38.W0 4A /r
- MVEX.512.66.0F38.W0 4B /r
- MVEX.512.66.0F38.W0 68 /r
- MVEX.512.66.0F38.W0 69 /r
- MVEX.512.66.0F38.W0 6A /r
- MVEX.512.66.0F38.W0 6B /r
- MVEX.512.66.0F38.W0 B0 /r /vsib
- MVEX.512.66.0F38.W0 B2 /r /vsib
- MVEX.512.66.0F38.W0 C0 /r /vsib
- MVEX.512.66.0F38.W0 D2 /r
- MVEX.512.66.0F38.W0 D6 /r
- MVEX.512.66.0F3A.W0 D0 /r ib
- MVEX.512.66.0F3A.W0 D1 /r ib
- MVEX.NDS.512.66.0F38.W0 54 /r

- MVEX.NDS.512.66.0F38.W0 56 /r
- MVEX.NDS.512.66.0F38.W0 57 /r
- MVEX.NDS.512.66.0F38.W0 67 /r
- MVEX.NDS.512.66.0F38.W0 70 /r
- MVEX.NDS.512.66.0F38.W0 71 /r
- MVEX.NDS.512.66.0F38.W0 72 /r
- MVEX.NDS.512.66.0F38.W0 73 /r
- MVEX.NDS.512.66.0F38.W0 94 /r
- MVEX.NDS.512.66.0F38.W0 CE /r
- MVEX.NDS.512.66.0F38.W0 CF /r
- MVEX.NDS.512.66.0F38.W1 94 /r
- MVEX.NDS.512.66.0F38.W1 CE /r
- VEX.128.F2.0F38.W0 F0 /r
- VEX.128.F2.0F38.W0 F1 /r
- VEX.128.F2.0F38.W1 F0 /r
- VEX.128.F2.0F38.W1 F1 /r
- VEX.128.F3.0F38.W0 F0 /r
- VEX.128.F3.0F38.W1 F0 /r

Appendix F

General Templates

In this Chapter all the general templates are described. Each instruction has one (at least) valid format, and each format matches with one of these templates.

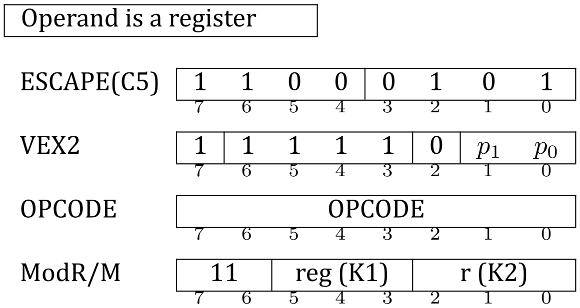
F.1 Mask Operation Templates

Mask m0 - Template

VMASKMask m0

Opcode	Instruction	Description
VEX.128	KOP k1, k2	Operate [mask k1 and] mask k2 [and store the result in k1]

Description



Mask m1 - Template

VMASKMask m1

Opcode	**Instruction**	**Description**
VEX.128	KOP r32/r64, k1, imm8	Move mask k1 into r32/r64 using imm8

Description

Operand is a register

	7	6	5	4	3	2	1	0
ESCAPE(C4)	1	1	0	0	0	1	0	0
VEX1	$!reg_3$	1	1	m_4	m_3	m_2	m_1	m_0
VEX2	W	1	1	1	1	L=0	p_1	p_0
OPCODE	OPCODE							
ModR/M	11		reg (reg)			r (K1)		
$\{IMM8\}$	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0

Mask m2 - Template

VMASKMask m2

Opcode	**Instruction**	**Description**

Description

Operand is a register

	7	6	5	4	3	2	1	0
ESCAPE(C4)	1	1	0	0	0	1	0	0
VEX1	$!reg_3$	1	1	m_4	m_3	m_2	m_1	m_0
VEX2	W	1	$!K1_2$	$!K1_1$	$!K1_0$	L=0	p_1	p_0
OPCODE	OPCODE							
ModR/M	11		reg (reg)			r (K2)		

Mask m3 - Template

VMASKMask m3

Opcode	**Instruction**	**Description**
VEX.128	KOP r32/r64, k1	Move mask k1 into r32/r64

Description

Operand is a register

ESCAPE(C5)	1	1	0	0	0	1	0	1
	7	6	5	4	3	2	1	0
VEX1	!reg_3	1	1	1	1	0	p_1	p_0
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	11		reg (reg)			r (K1)		
	7	6	5	4	3	2	1	0

Mask m4 - Template

VMASKMask m4

Opcode	**Instruction**	**Description**
VEX.128	KOP k1, r32/r64	Move r32/r64 into mask k1

Description

Operand is a register

C4 Version

	7	6	5	4	3	2	1	0
ESCAPE(C4)	1	1	0	0	0	1	0	0
VEX1	1	1	$!reg_3$	m_4	m_3	m_2	m_1	m_0
VEX2	W	1	1	1	1	0	p_1	p_0
OPCODE	OPCODE							
ModR/M	11		reg (K1)			r (reg)		

C5 Version

	7	6	5	4	3	2	1	0
ESCAPE(C5)	1	1	0	0	0	1	0	1
VEX1	1	1	1	1	1	0	p_1	p_0
OPCODE	OPCODE							
ModR/M	11		reg (K1)			r (reg)		

Mask m5 - Template

VMASKMask m5

Opcode	**Instruction**	**Description**
VEX.128	KOP k1, r32/r64, imm8	Move r32/r64 field into mask k1 using imm8

Description

Operand is a register

ESCAPE(C4)	1	1	0	0	0	1	0	0
	7	6	5	4	3	2	1	0
VEX1	1	1	$!reg_3$	m_4	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
VEX2	W	1	1	1	1	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	11		reg (K1)			r (reg)		
	7	6	5	4	3	2	1	0
$\{IMM8\}$	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
	7	6	5	4	3	2	1	0

F.2 Vector Operation Templates

Vector v0 - Template

VectorVector v0

Opcode	Instruction	Description
MVEX.512	VOP zmm1 {k1}, zmm2, S(zmm3/m_t)	Operate vector zmm2 and vector S(zmm3/m_t) [and vector zmm1] and store the result in zmm1, under write-mask k1
MVEX.512	VOP zmm1 {k1}, zmm2, S(zmm3/m_t), imm8	Operate vector zmm2 and vector S(zmm3/m_t) [and vector zmm1] and store the result in zmm1 using imm8, under write-mask k1

Description

Operand is a register

	7	6	5	4	3	2	1	0
ESCAPE(62)	0	1	1	0	0	0	1	0
MVEX1	$!Z1_3$	$!Z3_4$	$!Z3_3$	$!Z1_4$	m_3	m_2	m_1	m_0
MVEX2	W	$!Z2_3$	$!Z2_2$	$!Z2_1$	$!Z2_0$	L=0	p_1	p_0
MVEX3	EH	S_2	S_1	S_0	$!Z2_4$	$K1_2$	$K1_1$	$K1_0$
OPCODE	OPCODE							
ModR/M	11		reg (ZMM1)			r (ZMM3)		
$\{IMM8\}$	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0

Operand is a memory location

	7	6	5	4	3	2	1	0
ESCAPE(62)	0	1	1	0	0	0	1	0
MVEX1	$!Z1_3$	!X	!B	$!Z1_4$	m_3	m_2	m_1	m_0
MVEX2	W	$!Z2_3$	$!Z2_2$	$!Z2_1$	$!Z2_0$	L=0	p_1	p_0
MVEX3	EH	S_2	S_1	S_0	$!Z2_4$	$K1_2$	$K1_1$	$K1_0$
OPCODE	OPCODE							
ModR/M	mod		reg (ZMM1)			m (mt)		
$\{SIB\}$	SIB byte							
$\{DISPL\}$	Displacement (8*N/32)							
	31,8	.	.	.	.	.	.	0

$\{IMM8\}$	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
	7	6	5	4	3	2	1	0

Vector v1 - Template

VectorVector v1

Opcode	Instruction	Description
MVEX.512	VOP zmm1 {k1}, $S(m_t)$	Load/brodcast vector $S(m_t)$ into zmm1, under write-mask k1

Description

Operand is a memory location

ESCAPE(62)	0	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0
MVEX1	$!Z1_3$	!X	!B	$!Z1_4$	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
MVEX2	W	1	1	1	1	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
MVEX3	EH	S_2	S_1	S_0	1	$K1_2$	$K1_1$	$K1_0$
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	mod		reg (ZMM1)			m (mt)		
	7	6	5	4	3	2	1	0
$\{SIB\}$	SIB byte							
	7	6	5	4	3	2	1	0
$\{DISPL\}$	Displacement (8*N/32)							
	31,8	.	.	.	.	.	.	0

Vector v10 - Template

VectorVector v10

Opcode	Instruction	Description
MVEX.512	VOP zmm1 {k1}, S(zmm2/m_t)	Operate vector S(zmm2/m_t) and store the result in zmm1, under write-mask k1
MVEX.512	VOP zmm1 {k1}, S(zmm2/m_t), imm8	Operate vector S(zmm2/m_t) and store the result in zmm1 using imm8, under write-mask k1
MVEX.512	VOP zmm1 {k1}, S(zmm2/m_t)	Move vector S(zmm2/m_t) into zmm1, under write-mask k1

Description

Operand is a register

ESCAPE(62)	0	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0
MVEX1	1	$!Z24$	$!Z23$	1	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
MVEX2	W	$!Z1_3$	$!Z1_2$	$!Z1_1$	$!Z1_0$	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
MVEX3	EH	S_2	S_1	S_0	$!Z1_4$	$K1_2$	$K1_1$	$K1_0$
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	11		Op. Ext.			r (ZMM2)		
	7	6	5	4	3	2	1	0
$\{IMM8\}$	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
	7	6	5	4	3	2	1	0

Operand is a memory location

ESCAPE(62)	0	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0
MVEX1	1	!X	!B	1	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
MVEX2	W	$!Z1_3$	$!Z1_2$	$!Z1_1$	$!Z1_0$	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
MVEX3	EH	S_2	S_1	S_0	$!Z1_4$	$K1_2$	$K1_1$	$K1_0$
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	mod		Op. Ext.			m (mt)		
	7	6	5	4	3	2	1	0
$\{SIB\}$	SIB byte							
	7	6	5	4	3	2	1	0
$\{DISPL\}$	Displacement (8*N/32)							
	31,8	.	.	.	.	.	.	0

$\{IMM8\}$	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
	7	6	5	4	3	2	1	0

Vector v11 - Template

VectorVector v11

Opcode	**Instruction**	**Description**
MVEX.512	VOP zmm1 {k1}, zmm2, $S(m_t)$	Load/brodcast and OP vector $S(m_t)$ with zmm2 and write result into zmm1, under write-mask k1

Description

Operand is a memory location

ESCAPE(62)	0	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0
MVEX1	$!Z1_3$	!X	!B	$!Z1_4$	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
MVEX2	W	$!Z2_3$	$!Z2_2$	$!Z2_1$	$!Z2_0$	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
MVEX3	EH	S_2	S_1	S_0	$!Z2_4$	$K1_2$	$K1_1$	$K1_0$
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	mod		reg (ZMM1)			m (mt)		
	7	6	5	4	3	2	1	0
$\{SIB\}$	SIB byte							
	7	6	5	4	3	2	1	0
$\{DISPL\}$	Displacement (8*N/32)							
	31,8	.	.	.	.	.	.	0

Vector v2 - Template

VectorVector v2

Opcode	**Instruction**	**Description**
MVEX.512	VOP k2 {k1}, zmm2, S(zmm3/m_t)	Operate vector zmm2 and vector S(zmm3/m_t) and store the result in k2, under write-mask k1
MVEX.512	VOP k2 {k1}, zmm2, S(zmm3/m_t), imm8	Operate vector zmm2 and vector S(zmm3/m_t) and store the result in k2 using imm8, under write-mask k1

Description

Operand is a register

ESCAPE(62)	0	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0
MVEX1	1	$!Z2_4$	$!Z2_3$	1	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
MVEX2	W	$!Z1_3$	$!Z1_2$	$!Z1_1$	$!Z1_0$	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
MVEX3	EH	S_2	S_1	S_0	$!Z1_4$	$K1_2$	$K1_1$	$K1_0$
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	11		reg (K2)			r (ZMM2)		
	7	6	5	4	3	2	1	0
$\{IMM8\}$	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
	7	6	5	4	3	2	1	0

Operand is a memory location

ESCAPE(62)	0	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0
MVEX1	1	!X	!B	1	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
MVEX2	W	$!Z1_3$	$!Z1_2$	$!Z1_1$	$!Z1_0$	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
MVEX3	EH	S_2	S_1	S_0	$!Z1_4$	$K1_2$	$K1_1$	$K1_0$
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	mod		reg (K2)			m (mt)		
	7	6	5	4	3	2	1	0
$\{SIB\}$	SIB byte							
	7	6	5	4	3	2	1	0
$\{DISPL\}$	Displacement (8*N/32)							
	31,8	.	.	.	.	.	.	0
$\{IMM8\}$	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0

7	6	5	4	3	2	1	0

Vector v3 - Template

VectorVector v3

Opcode	Instruction	Description
MVEX.512	VOP m_t {k1}, D(zmm1)	Store vector D(zmm1) into m_t, under write-mask k1

Description

Operand is a memory location

ESCAPE(62)	0	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0
MVEX1	$!Z1_3$	!X	!B	$!Z1_4$	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
MVEX2	W	1	1	1	1	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
MVEX3	EH	S_2	S_1	S_0	1	$K1_2$	$K1_1$	$K1_0$
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	mod		reg (ZMM1)			m (mt)		
	7	6	5	4	3	2	1	0
$\{SIB\}$	SIB byte							
	7	6	5	4	3	2	1	0
$\{DISPL\}$	Displacement (8*N/32)							
	31,8	.	.	.	.	.	.	0

Vector v4 - Template

VectorVector v4

Opcode	Instruction	Description
MVEX.512	VOP zmm1 {k1}, zmm2/m_t	Operate vector zmm2/m_t and store the result in zmm1, under write-mask k1
MVEX.512	VOP zmm1 {k1}, zmm2/m_t, imm8	Operate vector zmm2/m_t and store the result in zmm1 using imm8, under write-mask k1

Description

Operand is a register

ESCAPE(62)	0	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0
MVEX1	$!Z1_3$	$!Z2_4$	$!Z2_3$	$!Z1_4$	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
MVEX2	W	1	1	1	1	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
MVEX3	EH	0	0	0	1	$K1_2$	$K1_1$	$K1_0$
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	11		reg (ZMM1)			r (ZMM2)		
	7	6	5	4	3	2	1	0
$\{IMM8\}$	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
	7	6	5	4	3	2	1	0

Operand is a memory location

ESCAPE(62)	0	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0
MVEX1	$!Z1_3$	!X	!B	$!Z1_4$	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
MVEX2	W	1	1	1	1	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
MVEX3	EH	0	0	0	1	$K1_2$	$K1_1$	$K1_0$
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	mod		reg (ZMM1)			m (mt)		
	7	6	5	4	3	2	1	0
$\{SIB\}$	SIB byte							
	7	6	5	4	3	2	1	0
$\{DISPL\}$	Displacement (8*N/32)							
	31,8	.	.	.	.	.	.	0
$\{IMM8\}$	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
	7	6	5	4	3	2	1	0

Vector v5 - Template

VectorVector v5

Opcode	**Instruction**	**Description**
MVEX.512	VOP zmm1 {k1}, S(zmm2/m_t)	Operate vector S(zmm2/m_t) and store the result in zmm1, under write-mask k1
MVEX.512	VOP zmm1 {k1}, S(zmm2/m_t), imm8	Operate vector S(zmm2/m_t) and store the result in zmm1 using imm8, under write-mask k1
MVEX.512	VOP zmm1 {k1}, S(zmm2/m_t)	Move vector S(zmm2/m_t) into zmm1, under write-mask k1

Description

Operand is a register

	7	6	5	4	3	2	1	0
ESCAPE(62)	0	1	1	0	0	0	1	0
MVEX1	$!Z1_3$	$!Z2_4$	$!Z2_3$	$!Z1_4$	m_3	m_2	m_1	m_0
MVEX2	W	1	1	1	1	L=0	p_1	p_0
MVEX3	EH	S_2	S_1	S_0	1	$K1_2$	$K1_1$	$K1_0$
OPCODE	OPCODE							
ModR/M	11		reg (ZMM1)			r (ZMM2)		
$\{IMM8\}$	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0

Operand is a memory location

	7	6	5	4	3	2	1	0
ESCAPE(62)	0	1	1	0	0	0	1	0
MVEX1	$!Z1_3$	!X	!B	$!Z1_4$	m_3	m_2	m_1	m_0
MVEX2	W	1	1	1	1	L=0	p_1	p_0
MVEX3	EH	S_2	S_1	S_0	1	$K1_2$	$K1_1$	$K1_0$
OPCODE	OPCODE							
ModR/M	mod		reg (ZMM1)			m (mt)		
$\{SIB\}$	SIB byte							
$\{DISPL\}$	Displacement (8*N/32)							
	31,8	.	.	.	.	.	.	0

$\{IMM8\}$	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
	7	6	5	4	3	2	1	0

Vector v6 - Template

VectorVector v6

Opcode	**Instruction**	**Description**
MVEX.512	VOP zmm1 {k1}, $S(mv_t)$	Gather sparse vector $S(mv_t)$ into zmm1, using completion mask k1
MVEX.512	VOP mv_t {k1}, D(zmm1)	Scatter vector D(zmm1) into sparse vector mv_t, using completion mask k1

Description

Operand is a memory location

ESCAPE(62)	0	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0
MVEX1	$!Z1_3$	$!X_3$	$!B_3$	$!Z1_4$	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
MVEX2	W	1	1	1	1	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
MVEX3	EH	S_2	S_1	S_0	$!X_4$	$K1_2$	$K1_1$	$K1_0$
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	mod		reg (ZMM1)			m= 100		
	7	6	5	4	3	2	1	0
$VSIB$	SS_1	SS_0	Index(X)			Base(B)		
	7	6	5	4	3	2	1	0
$\{DISPL\}$	Displacement (8*N/32)							
	31,8	.	.	.	.	.	.	0
$\{IMM8\}$	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
	7	6	5	4	3	2	1	0

Vector v7 - Template

VectorVector v7

Opcode	**Instruction**	**Description**
MVEX.512	VOP zmm1 {k1}, k2, S(zmm3/m_t)	Operate mask k2 and vector S(zmm3/m_t) [and vector zmm1], and store the result in zmm1, under write-mask k1

Description

Operand is a register

ESCAPE(62)	0	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0
MVEX1	$!Z1_3$	$!Z3_4$	$!Z3_3$	$!Z1_4$	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
MVEX2	W	1	$!K2_2$	$!K2_1$	$!K2_0$	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
MVEX3	EH	S_2	S_1	S_0	1	$K1_2$	$K1_1$	$K1_0$
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	11		reg (ZMM1)			r (ZMM3)		
	7	6	5	4	3	2	1	0

Operand is a memory location

ESCAPE(62)	0	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0
MVEX1	$!Z1_3$	!X	!B	$!Z1_4$	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
MVEX2	W	1	$!K2_2$	$!K2_1$	$!K2_0$	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
MVEX3	EH	S_2	S_1	S_0	1	$K1_2$	$K1_1$	$K1_0$
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	mod		reg (ZMM1)			m (mt)		
	7	6	5	4	3	2	1	0
$\{SIB\}$	SIB byte							
	7	6	5	4	3	2	1	0
$\{DISPL\}$	Displacement (8*N/32)							
	31,8	.	.	.	.	.	.	0

Vector v8 - Template

VectorVector v8

Opcode	Instruction	Description
MVEX.512	VOP zmm1 {k1}, zmm2, zmm3/m_t	Operate vector zmm2 and vector zmm3/m_t [and vector zmm1] and store the result in zmm1, under write-mask k1
MVEX.512	VOP zmm1 {k1}, zmm2, zmm3/m_t, imm8	Operate vector zmm2 and vector zmm3/m_t [and vector zmm1] and store the result in zmm1 using imm8, under write-mask k1

Description

Operand is a register

ESCAPE(62)	0	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0
MVEX1	$!Z1_3$	$!Z3_4$	$!Z3_3$	$!Z1_4$	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
MVEX2	W	$!Z2_3$	$!Z2_2$	$!Z2_1$	$!Z2_0$	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
MVEX3	EH	0	0	0	$!Z2_4$	$K1_2$	$K1_1$	$K1_0$
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	11		reg (ZMM1)			r (ZMM3)		
	7	6	5	4	3	2	1	0
$\{IMM8\}$	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
	7	6	5	4	3	2	1	0

Operand is a memory location

ESCAPE(62)	0	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0
MVEX1	$!Z1_3$	!X	!B	$!Z1_4$	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
MVEX2	W	$!Z2_3$	$!Z2_2$	$!Z2_1$	$!Z2_0$	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
MVEX3	EH	0	0	0	$!Z2_4$	$K1_2$	$K1_1$	$K1_0$
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	mod		reg (ZMM1)			m (mt)		
	7	6	5	4	3	2	1	0
$\{SIB\}$	SIB byte							
	7	6	5	4	3	2	1	0
$\{DISPL\}$	Displacement (8*N/32)							
	31,8	.	.	.	.	.	.	0

$\{IMM8\}$	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
	7	6	5	4	3	2	1	0

Vector v9 - Template

VectorVector v9

Opcode	**Instruction**	**Description**
MVEX.512	VOP $S(mv_t)$ {k1}	Prefetch sparse vector $S(mv_t)$, under write-mask k1

Description

Operand is a memory location

ESCAPE(62)	0	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0
MVEX1	1	$!X_3$	$!B_3$	1	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
MVEX2	W	1	1	1	1	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
MVEX3	EH	S_2	S_1	S_0	$!X_4$	$K1_2$	$K1_1$	$K1_0$
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	mod		Op. Ext.			m= 100		
	7	6	5	4	3	2	1	0
$VSIB$	SS_1	SS_0	Index(X)			Base(B)		
	7	6	5	4	3	2	1	0
$\{DISPL\}$	Displacement (8*N/32)							
	31,8	.	.	.	.	.	.	0

F.3 Scalar Operation Templates

Scalar s0 - Template

scalarScalar s0

Opcode	Instruction	Description
0F/0F38/0F3A	OP r16, r16/m16	Operate [r16 and] r16/m16, leaving the result in r16
0F/0F38/0F3A	OP r32, r32/m32	Operate [r32 and] r32/m32, leaving the result in r32
REX.W 0F/0F38/0F3A	OP r64, r64/m64	Operate [r64 and] r64/m64, leaving the result in r64

Description

Operand is a register

C4 Version

	7	6	5	4	3	2	1	0
ESCAPE(C4)	1	1	0	0	0	1	0	0
VEX1	$!dst_3$	1	$!src_3$	m_4	m_3	m_2	m_1	m_0
VEX2	W	1	1	1	1	L=0	p_1	p_0
OPCODE	OPCODE							
ModR/M	11		reg (dst)			r (src)		

C5 Version

	7	6	5	4	3	2	1	0
ESCAPE(C5)	1	1	0	0	0	1	0	1
VEX2	$!dst_3$	1	1	1	1	L=0	p_1	p_0
OPCODE	OPCODE							
ModR/M	11		reg (dst)			r (src)		

Scalar s1 - Template

scalarScalar s1

Opcode	**Instruction**	**Description**
VEX.128	OP m_t	Prefetch/Evict m_t memory location

Description

Operand is a memory location

C4 Version

ESCAPE(C4)	1	1	0	0	0	1	0	0
	7	6	5	4	3	2	1	0
VEX1	1	!X	!B	m_4	m_3	m_2	m_1	m_0
	7	6	5	4	3	2	1	0
VEX2	W	1	1	1	1	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	mod		Op. Ext.			m (mt)		
	7	6	5	4	3	2	1	0
$\{SIB\}$	SIB byte							
	7	6	5	4	3	2	1	0
$\{DISPL\}$	Displacement (8/32)							
	31,8	.	.	.	.	.	.	0

C5 Version

ESCAPE(C5)	1	1	0	0	0	1	0	1
	7	6	5	4	3	2	1	0
VEX2	1	1	1	1	1	L=0	p_1	p_0
	7	6	5	4	3	2	1	0
OPCODE	OPCODE							
	7	6	5	4	3	2	1	0
ModR/M	mod		Op. Ext.			m (mt)		
	7	6	5	4	3	2	1	0
$\{SIB\}$	SIB byte							
	7	6	5	4	3	2	1	0
$\{DISPL\}$	Displacement (8/32)							
	31,8	.	.	.	.	.	.	0