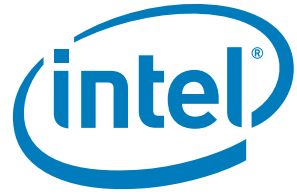


Intel® Advanced Vector Extensions Programming Reference

JUNE 2010

319433-007

CONTENTS

PAGE

CHAPTER 1
INTEL® ADVANCED VECTOR EXTENSIONS

CHAPTER 2
APPLICATION PROGRAMMING MODEL

CHAPTER 3
SYSTEM PROGRAMMING MODEL

CHAPTER 4
INSTRUCTION FORMAT

CHAPTER 5
INSTRUCTION SET REFERENCE

CHAPTER 6
INSTRUCTION SET REFERENCE - FMA

CHAPTER 7
POST-32NM PROCESSOR INSTRUCTIONS

APPENDIX A
INSTRUCTION SUMMARY

APPENDIX B
INSTRUCTION OPCODE MAP

TABLES

PAGE

FIGURES

PAGE

CHAPTER 1
INTEL® ADVANCED VECTOR EXTENSIONS

1.1 ABOUT THIS DOCUMENT

This document describes the software programming interfaces of several vector SIMD instruction extensions of the Intel® 64 architecture that will be introduced starting with Intel 64 processors built on 32nm process technology. The instruction set extensions covered in this document, with respect to availability in different processor generations, is referred to by the following categories:

- General-purpose encryption and AES: 128-bit SIMD extensions targeted to accelerate high-speed block encryption and cryptographic processing using the Advanced Encryption Standard.
- Intel® Advanced Vector Extensions (AVX) introduces 256-bit vector processing capability and includes two components to be introduced on Intel processor generations built from 32nm process and beyond:
  - The first generation Intel AVX provides 256-bit SIMD register support, 256-bit vector floating-point instructions, enhancements to 128-bit SIMD instructions, support for three and four operand syntax.
  - FMA is a future extension of Intel AVX, FMA provides floating-point, fused multiply-add instructions supporting 256-bit and 128-bit SIMD vectors.

Chapter 1 provides an overview of these instruction set extensions. Chapter 2 describes the application programming environment. Chapter 3 describes system programming requirements needed to support 256-bit registers. Chapter 4 describes the architectural extensions of Intel 64 instruction encoding format that support 256-bit registers, three and four operand syntax. Chapter 5 provides detailed instruction reference for AVX and encryption/AES instructions. Chapter 6 provides detailed instruction reference for FMA instructions.

1.2 OVERVIEW

Intel® Advanced Vector Extensions extend beyond the capabilities and programming environment over those of multiple generations of Streaming SIMD Extensions. Intel AVX address the continued need for vector floating-point performance in mainstream scientific and engineering numerical applications, visual processing, recognition, data-mining/synthesis, gaming, physics, cryptography and other areas of applications. Intel AVX is designed to facilitate efficient implementation by wide spectrum of software architectures of varying degrees of thread parallelism, and data vector lengths. Intel AVX offers the following benefits:

- efficient building blocks for applications targeted across all segments of computing platforms.

- significant increase in floating-point performance density with good power efficiency over previous generations of 128-bit SIMD instruction set extensions,
- scalable performance with multi-core processor capability.

Intel AVX also establishes a foundation for future evolution in both instruction set functionality and vector lengths by introducing an efficient instruction encoding scheme, three and four operand instruction syntax, supporting load and store masking, etc.

Intel Advanced Vector Extensions offers comprehensive architectural enhancements and functional enhancements in arithmetic as well as data processing primitives. Section 1.3 summarizes the architectural enhancement of AVX. Functional overview of AVX and FMA instructions are summarized in Section 1.4. General-purpose encryption and AES instructions follow the existing architecture of 128-bit SIMD instruction sets like SSE4 and its predecessors, Section 1.5 provides a short summary.

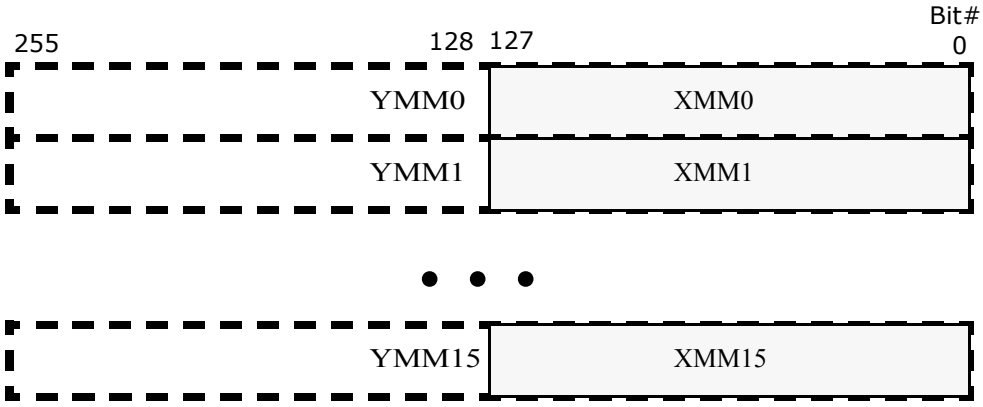
1.3 INTEL® ADVANCED VECTOR EXTENSIONS ARCHITECTURE OVERVIEW

Intel AVX has many similarities to the SSE and double-precision floating-point portions of SSE2. However, Intel AVX introduces the following architectural enhancements:

- Support for 256-bit wide vectors and SIMD register set. 256-bit register state is managed by Operating System using XSAVE/XRSTOR instructions introduced in 45 nm Intel 64 processors (see *IA-32 Intel® Architecture Software Developer's Manual, Volumes 2B and 3A*).
- Instruction syntax support for generalized three-operand syntax to improve instruction programming flexibility and efficient encoding of new instruction extensions.
- Enhancement of legacy 128-bit SIMD instruction extensions to support three-operand syntax and to simplify compiler vectorization of high-level language expressions.
- Instruction encoding format using a new prefix (referred to as VEX) to provide compact, efficient encoding for three-operand syntax, vector lengths, compaction of SIMD prefixes and REX functionality.
- FMA extensions and enhanced floating-point compare instructions add support for IEEE-754-2008 standard.

1.3.1 256-Bit Wide SIMD Register Support

Intel AVX introduces support for 256-bit wide SIMD registers (YMM0-YMM7 in operating modes that are 32-bit or less, YMM0-YMM15 in 64-bit mode). The lower 128-bits of the YMM registers are aliased to the respective 128-bit XMM registers.



1.3.2 Instruction Syntax Enhancements

Intel AVX employs an instruction encoding scheme using a new prefix (known as "VEX" prefix). Instruction encoding using the VEX prefix can directly encode a register operand within the VEX prefix. This support two new instruction syntax in Intel 64 architecture:

- A non-destructive operand (in a three-operand instruction syntax): The non-destructive source reduces the number of registers, register-register copies and explicit load operations required in typical SSE loops, reduces code size, and improves micro-fusion opportunities.
- A third source operand (in a four-operand instruction syntax) via the upper 4 bits in an 8-bit immediate field. Support for the third source operand is defined for selected instructions (e.g. VBLENDVPD, VBLENDVPS, PBLENDVB).

Two-operand instruction syntax previously expressed as

```
ADDPS xmm1, xmm2/m128
```

now can be expressed in three-operand syntax as

```
VADDPS xmm1, xmm2, xmm3/m128
```

In four-operand syntax, the extra register operand is encoded in the immediate byte.

Note SIMD instructions supporting three-operand syntax but processing only 128-bits of data are considered part of the 256-bit SIMD instruction set extensions of AVX, because bits 255:128 of the destination register are zeroed by the processor.

1.3.3 VEX Prefix Instruction Encoding Support

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

- Direct encoding of a register operand within VEX. This provides instruction syntax support for non-destructive source operand.
- Efficient encoding of instruction syntax operating on 128-bit and 256-bit register sets.
- Compaction of REX prefix functionality: The equivalent functionality of the REX prefix is encoded within VEX.
- Compaction of SIMD prefix functionality and escape byte encoding: The functionality of SIMD prefix (66H, F2H, F3H) on opcode is equivalent to an opcode extension field to introduce new processing primitives. This functionality is replaced by a more compact representation of opcode extension within the VEX prefix. Similarly, the functionality of the escape opcode byte (0FH) and two-byte escape (0F38H, 0F3AH) are also compacted within the VEX prefix encoding.
- Most VEX-encoded SIMD numeric and data processing instruction semantics with memory operand have relaxed memory alignment requirements than instructions encoded using SIMD prefixes (see Section 2.5).

VEX prefix encoding applies to SIMD instructions operating on YMM registers, XMM registers, and in some cases with a general-purpose register as one of the operand. VEX prefix is not supported for instructions operating on MMX or x87 registers. Details of VEX prefix and instruction encoding are discussed in Chapter 4.

1.4 FUNCTIONAL OVERVIEW

Intel AVX and FMA provide comprehensive functional improvements over previous generations of SIMD instruction extensions. The functional improvements include:

- 256-bit floating-point arithmetic primitives: AVX enhances existing 128-bit floating-point arithmetic instructions with 256-bit capabilities for floating-point processing. FMA provides additional set of 256-bit floating-point processing capabilities with a rich set of fused-multiply-add and fused multiply-subtract primitives.
- Enhancements for flexible SIMD data movements: AVX provides a number of new data movement primitives to enable efficient SIMD programming in relation to loading non-unit-strided data into SIMD registers, intra-register SIMD data manipulation, conditional expression and branch handling, etc. Enhancements

for SIMD data movement primitives cover 256-bit and 128-bit vector floating-point data, and across 128-bit integer SIMD data processing using VEX-encoded instructions.

Several key categories of functional improvements in AVX and FMA are summarized in the following subsections.

1.4.1 256-bit Floating-Point Arithmetic Processing Enhancements

Intel AVX provides 35 256-bit floating-point arithmetic instructions. The arithmetic operations cover add, subtract, multiply, divide, square-root, compare, max, min, round, etc., on single-precision and double-precision floating-point data.

The enhancement in AVX on floating-point compare operation provides 32 conditional predicates to improve programming flexibility in evaluating conditional expressions.

FMA provides 36 256-bit floating-point instructions to perform computation on 256-bit vectors. The arithmetic operations cover fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and multiply-subtract.

1.4.2 256-bit Non-Arithmetic Instruction Enhancements

Intel AVX provides new primitives for handling data movement within 256-bit floating-point vectors and promotes many 128-bit floating data processing instructions to handle 256-bit floating-point vectors.

AVX includes 33 256-bit data processing instructions that are promoted from previous generations of SIMD instruction extensions, ranging from logical, blend, convert, test, unpacking, shuffling, load and stores.

AVX introduces 19new data processing instructions that operate on 256-bit vectors. These new primitives cover the following operations:

- Non-unit-strided fetching of SIMD data. AVX provides several flexible SIMD floating-point data fetching primitives:
  - broadcast of single or multiple data elements into a 256-bit destination,
  - masked move primitives to load or store SIMD data elements conditionally,
- Intra-register manipulation of SIMD data elements. AVX provides several flexible SIMD floating-point data manipulation primitives:
  - insert/extract multiple SIMD floating-point data elements to/from 256-bit SIMD registers
  - permute primitives to facilitate efficient manipulation of floating-point data elements in 256-bit SIMD registers
- Branch handling. AVX provides several primitives to enable handling of branches in SIMD programming:

— new variable blend instructions supports four-operand syntax with non-destructive source syntax. This is more flexible than the equivalent SSE4 instruction syntax which uses the XMM0 register as the implied mask for blend selection.

— Packed TEST instructions for floating-point data.

1.4.3 Arithmetic Primitives for 128-bit Vector and Scalar processing

Intel AVX provides 131 128-bit numeric processing instructions that employ VEX-prefix encoding. These VEX-encoded instructions generally provide the same functionality over instructions operating on XMM register that are encoded using SIMD prefixes. The 128-bit numeric processing instructions in AVX cover floating-point and integer data processing; across 128-bit vector and scalar processing.

The enhancement in AVX on 128-bit floating-point compare operation provides 32 conditional predicates to improve programming flexibility in evaluating conditional expressions. This contrasts with floating-point SIMD compare instructions in SSE and SSE2 supporting only 8 conditional predicates.

FMA provides 60 128-bit floating-point instructions to process 128-bit vector and scalar data. The arithmetic operations cover fused multiply-add, fused multiply-subtract, signed-reversed multiply on fused multiply-add and multiply-subtract.

1.4.4 Non-Arithmetic Primitives for 128-bit Vector and Scalar Processing

Intel AVX provides 126 data processing instructions that employ VEX-prefix encoding. These VEX-encoded instructions generally provide the same functionality over instructions operating on XMM register that are encoded using SIMD prefixes. The 128-bit data processing instructions in AVX cover floating-point and integer data movement primitives.

Additional enhancements in AVX on 128-bit data processing primitives include 16 new instructions with the following capabilities:

- Non-unit-strided fetching of SIMD data. AVX provides several flexible SIMD floating-point data fetching primitives:
  — broadcast of single data element into a 128-bit destination,
  — masked move primitives to load or store SIMD data elements conditionally,
- Intra-register manipulation of SIMD data elements. AVX provides several flexible SIMD floating-point data manipulation primitives:
  — permute primitives to facilitate efficient manipulation of floating-point data elements in 128-bit SIMD registers
- Branch handling. AVX provides several primitives to enable handling of branches in SIMD programming:

— new variable blend instructions supports four-operand syntax with non-destructive source syntax. Branching conditions dependent on floating-point data or integer data can benefit from Intel AVX. This is more flexible than non-VEX encoded instruction syntax that uses the XMM0 register as implied mask for blend selection. While variable blend with implied XMM0 syntax is supported in SSE4 using SIMD prefix encoding, VEX-encoded 128-bit variable blend instructions only support the more flexible four-operand syntax.

— Packed TEST instructions for floating-point data.

1.5 GENERAL ENCRYPTION AND CRYPTOGRAPHIC PROCESSING

Intel 64 processors using 32nm processing technology introduces several primitives targeted to accelerate general-purpose block encryption and cryptographic functions using the Advanced Encryption Standard (AES) of block cipher encryption and decryption on 128-bit blocks.

General-purpose block encryption primitives are provided by PCLMULQDQ instruction, which can perform carry-less multiplication for two binary numbers up to 64-bit wide.

AES encryption involves processing 128-bit input data (plaintext) through a finite number of iterative operation, referred to as "AES round", into a 128-bit encrypted block (ciphertext). Decryption follows the reverse direction of iterative operation using the "equivalent inverse cipher" instead of the "inverse cipher".

The cryptographic processing at each round involves two input data, one is the "state", the other is the "round key". Each round uses a different "round key". The round keys are derived from the cipher key using a "key schedule" algorithm. The "key schedule" algorithm is independent of the data processing of encryption/decryption, and can be carried out independently from the encryption/decryption phase.

The AES standard supports cipher key of sizes 128, 192, and 256 bits. The respective cipher key sizes correspond to 10, 12, and 14 rounds of iteration.

The AES extensions provide two primitives to accelerate AES rounds on encryption, two primitives for AES rounds on decryption using the equivalent inverse cipher, and two instructions to support the AES key expansion procedure.

This page was intentionally left blank.

§

CHAPTER 2
APPLICATION PROGRAMMING MODEL

The application programming model for Intel AVX, FMA, AES and encryption primitives extend from that of Streaming SIMD Extensions (SSE) and is summarized as follows:

- The AES extensions and carry-less multiplication instruction (PCLMULQDQ) follow the same programming model as SSE, SSE2, SSE3, SSSE3, and SSE4 (see *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). The detection of AES and PCLMULQDQ is described in Section 2.1.
- The AVX and FMA extensions follow a programming model analogous to that of SSE with minor differences. This is described in Section 2.1 through Section 2.8. Note however that the OS support and detection process has changed considerably.
- The numeric exception behavior of FMA is similar to previous generations of SIMD floating-point instructions. The specific details are described in Section 2.3.

CPUID instruction details for detecting AVX, FMA, AES, PCLMULQDQ are described in Section 2.9.

2.1 DETECTION OF PCLMULQDQ AND AES INSTRUCTIONS

Before an application attempts to use the following AES instructions: AESDEC/AESDECLAST/AESENC/AESENCLAST/AESIMC/AESKEYGENASSIST, it must check that the processor supports the AES extensions. AES extensions is supported if CPUID.01H:ECX.AES[bit 25] = 1.

Prior to using PCLMULQDQ instruction, application must check if CPUID.01H:ECX.PCLMULQDQ[bit 1] = 1.

Operating systems that support handling SSE state will also support applications that use AES extensions and PCLMULQDQ instruction. This is the same requirement for SSE2, SSE3, SSSE3, and SSE4.

2.2 DETECTION OF AVX AND FMA INSTRUCTIONS

AVX and FMA operate on the 256-bit YMM register state. System software requirements to support YMM state is described in Chapter 3.

Application detection of new instruction extensions operating on the YMM state follows the general procedural flow in Figure 2-1.

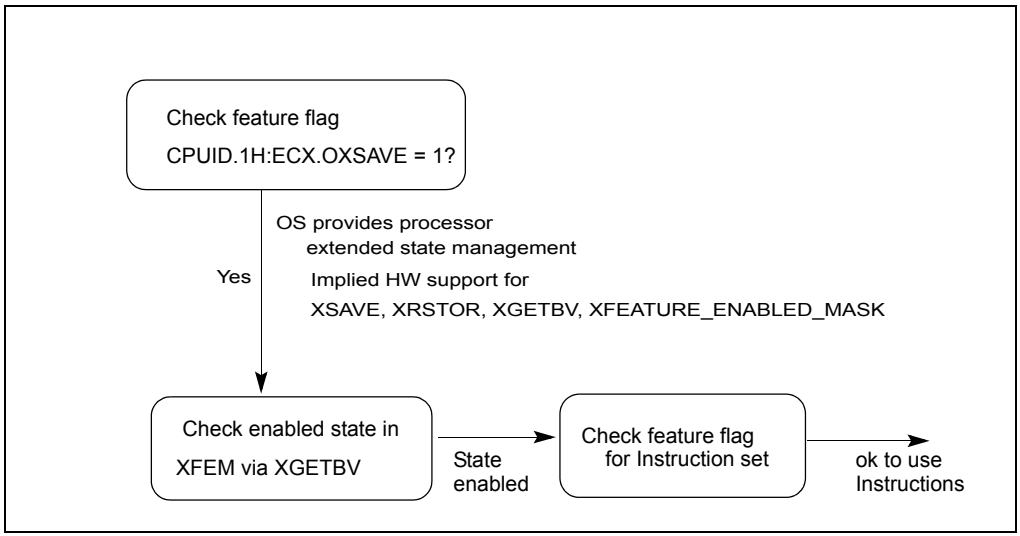


Figure 2-1. General Procedural Flow of Application Detection of AVX

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

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

2) Issue XGETBV and verify that XFEATURE_ENABLED_MASK[2:1] = '11b' (XMM state and YMM state are enabled by OS).

3) detect CPUID.1:ECX.AVX[bit 28] = 1 (AVX instructions supported).

(Step 3 can be done in any order relative to 1 and 2)

The following pseudocode illustrates this recommended application AVX detection process:

```
-----------------------------------------------------------------------------
INT supports_AVX()
{       ; result in eax
        mov eax, 1
        cpuid
```

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

```
        and ecx, 018000000H
        cmp ecx, 018000000H; check both OSXSAVE and AVX feature flags
        jne not_supported
        ; processor supports AVX instructions and XGETBV is enabled by OS
        mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
        XGETBV; result in EDX:EAX
        and eax, 06H
        cmp eax, 06H; check OS has enabled both XMM and YMM state support
        jne not_supported
        mov eax, 1
        jmp done
        NOT_SUPPORTED:
        mov eax, 0
        done:
}
```

Note: It is unwise for an application to rely exclusively on CPUID.1:ECX.AVX[bit 28] or at all on CPUID.1:ECX.XSAVE[bit 26]: These indicate hardware support but not operating system support. If YMM state management is not enabled by an operating systems, AVX instructions will #UD regardless of CPUID.1:ECX.AVX[bit 28]. "CPUID.1:ECX.XSAVE[bit 26] = 1" does not guarantee the OS actually uses the XSAVE process for state management.

These steps above also apply to enhanced 128-bit SIMD floating-pointing instructions in AVX (using VEX prefix-encoding) that operate on the YMM states. Application detection of VEX-encoded AES is described in Section 2.2.2.

2.2.1 Detection of FMA

Hardware support for FMA is indicated by CPUID.1:ECX.FMA[bit 12]=1.

Application Software must identify that hardware supports AVX as explained in Section 2.2, after that it must also detect support for FMA by CPUID.1:ECX.FMA[bit 12]. The recommended pseudocode sequence for detection of FMA is:

```
INT supports_fma()
{       ; result in eax
        mov eax, 1
        cpuid
        and ecx, 018001000H
```

```
        cmp ecx, 018001000H; check OSXSAVE, AVX, FMA feature flags
        jne not_supported
        ; processor supports AVX,FMA instructions and XGETBV is enabled by OS
        mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
        XGETBV; result in EDX:EAX
        and eax, 06H
        cmp eax, 06H; check OS has enabled both XMM and YMM state support
        jne not_supported
        mov eax, 1
        jmp done
        NOT_SUPPORTED:
        mov eax, 0
        done:
}
```

Note that FMA comprises of 256-bit and 128-bit SIMD instructions operating on YMM states.

2.2.2 Detection of VEX-Encoded AES and VPCLMULQDQ

VAESDEC/VAESDECLAST/VAESENC/VAESENCLAST/VAESIMC/VAESKEYGENASSIST instructions operate on YMM states. The detection sequence must combine checking for CPUID.1:ECX.AES[bit 25] = 1 and the sequence for detection application support for AVX.

Similarly, the detection sequence for VPCLMULQDQ must combine checking for CPUID.1:ECX.PCLMULQDQ[bit 1] = 1 and the sequence for detection application support for AVX.

This is shown in the pseudocode:

```
INT supports_VAES()
{       ; result in eax
        mov eax, 1
        cpuid
        and ecx, 01A000000H
        cmp ecx, 01A000000H; check OSXSAVE, AVX and AES feature flags
        jne not_supported
```

```
        ; processor supports AVX and VEX.128-encoded AES instructions and XGETBV
is enabled by OS
        mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
        XGETBV; result in EDX:EAX
        and eax, 06H
        cmp eax, 06H; check OS has enabled both XMM and YMM state support
        jne not_supported
        mov eax, 1
        jmp done
        NOT_SUPPORTED:
        mov eax, 0
        done:
}

INT supports_VPCLMULQDQ()
{       ; result in eax
        mov eax, 1
        cpuid
        and ecx, 018000002H
        cmp ecx, 018000002H; check OSXSAVE, AVX and PCLMULQDQ feature flags
         jne not_supported
        ; processor supports AVX and VPCLMULQDQ instructions and XGETBV is
enabled by OS
        mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
        XGETBV; result in EDX:EAX
        and eax, 06H
        cmp eax, 06H; check OS has enabled both XMM and YMM state support
        jne not_supported
        mov eax, 1
        jmp done
        NOT_SUPPORTED:
        mov eax, 0
        done:
}
```

2.3 FUSED-MULTIPLY-ADD (FMA) NUMERIC BEHAVIOR

FMA instructions can perform fused-multiply-add operations (including fused-multiply-subtract, and other varieties) on packed and scalar data elements in the instruction operands. FMA instruction provide separate instructions to handle different types of arithmetic operations on the three source operands.

FMA instruction syntax is defined using three source operands and the first source operand is updated based on the result of the arithmetic operations of the data elements of 128-bit or 256-bit operands, i.e. The first source operand is also the destination operand.

The arithmetic FMA operation performed in an FMA instruction takes one of several forms, r=(x*y)+z, r=(x*y)-z, r=-(x*y)+z, or r=-(x*y)-z. Packed FMA instructions can perform eight single-precision FMA operations or four double-precision FMA operations with 256-bit vectors.

Scalar FMA instructions only perform one arithmetic operation on the low order data element. The content of the rest of the data elements in the lower 128-bits of the destination operand is preserved. the upper 128bits of the destination operand are filled with zero.

An arithmetic FMA operation of the form, r=(x*y)+z, takes two IEEE-754-2008 single (double) precision values and multiplies them to form an infinite precision intermediate value. This intermediate value is added to a third single (double) precision value (also at infinite precision) and rounded to produce a single (double) precision result.

Table 2-2 describes the numerical behavior of the FMA operation, r=(x*y)+z, r=(x*y)-z, r=-(x*y)+z, r=-(x*y)-z for various input values. The input values can be 0, finite non-zero (F in Table 2-2), infinity of either sign (INF in Table 2-2), positive infinity (+INF in Table 2-2), negative infinity (-INF in Table 2-2), or NaN (including QNaN or SNaN). If any one of the input values is a NAN, the result of FMA operation, r, may be a quietized NAN. The result can be either Q(x), Q(y), or Q(z), see Table 2-2. If x is a NaN, then:

- Q(x) = x if x is QNaN or
- Q(x) = the quietized NaN obtained from x if x is SNaN

The notation for output value in Table 2-2 are:

- "+INF": positive infinity, "-INF": negative infinity. When the result depends on a conditional expression, both values are listed in the result column and the condition is described in the comment column.
- QNaNIndefinite represents the QNaN which has the sign bit equal to 1, the second most significand field equal to 1, and the remaining significand field bits equal to 0.

- The summation or subtraction of 0s or identical values in FMA operation can lead to the following situations shown in Table 2-1
- If the FMA computation represents an invalid operation (e.g. when adding two INF with opposite signs)), the invalid exception is signaled, and the MXCSR.IE flag is set.

Table 2-1. Rounding behavior of Zero Result in FMA Operation

x*y	z	(x*y) + z	(x*y) - z	- (x*y) + z	- (x*y) - z
(+0)	(+0)	+0 in all rounding modes	- 0 when rounding down, and +0 otherwise	- 0 when rounding down, and +0 otherwise	- 0 in all rounding modes
(+0)	(-0)	- 0 when rounding down, and +0 otherwise	+0 in all rounding modes	- 0 in all rounding modes	- 0 when rounding down, and +0 otherwise
(-0)	(+0)	- 0 when rounding down, and +0 otherwise	- 0 in all rounding modes	+ 0 in all rounding modes	- 0 when rounding down, and +0 otherwise
(-0)	(-0)	- 0 in all rounding modes	- 0 when rounding down, and +0 otherwise	- 0 when rounding down, and +0 otherwise	+ 0 in all rounding modes
F	-F	- 0 when rounding down, and +0 otherwise	2*F	-2*F	- 0 when rounding down, and +0 otherwise
F	F	2*F	- 0 when rounding down, and +0 otherwise	- 0 when rounding down, and +0 otherwise	-2*F

Table 2-2. FMA Numeric Behavior

x	y	z	r=(x*y) +z	r=(x*y) - z	r = -(x*y)+z	r= -(x*y)-z	Comment
NaN	0, F, INF, NaN	0, F, INF, NaN	Q(x)	Q(x)	Q(x)	Q(x)	Signal invalid exception if x or y or z is SNaN
0, F, INF	NaN	0, F, INF, NaN	Q(y)	Q(y)	Q(y)	Q(y)	Signal invalid exception if y or z is SNaN
0, F, INF	0, F, INF	NaN	Q(z)	Q(z)	Q(z)	Q(z)	Signal invalid exception if z is SNaN
INF	F, INF	+INF	+INF	QNaNIndefinite	QNaNIndefinite	-INF	if x*y and z have the same sign
			QNaNIndefinite	-INF	+INF	QNaNIndefinite	if x*y and z have opposite signs
INF	F, INF	-INF	-INF	QNaNIndefinite	QNaNIndefinite	+INF	if x*y and z have the same sign
			QNaNIndefinite	+INF	-INF	QNaNIndefinite	if x*y and z have opposite signs
INF	F, INF	0, F	+INF	+INF	-INF	-INF	if x and y have the same sign
			-INF	-INF	+INF	+INF	if x and y have opposite signs
INF	0	0, F, INF	QNaNIndefinite	QNaNIndefinite	QNaNIndefinite	QNaNIndefinite	Signal invalid exception
0	INF	0, F, INF	QNaNIndefinite	QNaNIndefinite	QNaNIndefinite	QNaNIndefinite	Signal invalid exception
F	INF	+INF	+INF	QNaNIndefinite	QNaNIndefinite	-INF	if x*y and z have the same sign
			QNaNIndefinite	-INF	+INF	QNaNIndefinite	if x*y and z have opposite signs
F	INF	-INF	-INF	QNaNIndefinite	QNaNIndefinite	+INF	if x*y and z have the same sign
			QNaNIndefinite	+INF	-INF	QNaNIndefinite	if x*y and z have opposite signs
F	INF	0,F	+INF	+INF	-INF	-INF	if x * y > 0
			-INF	-INF	+INF	+INF	if x * y < 0
0,F	0,F	INF	+INF	-INF	+INF	-INF	if z > 0
			-INF	+INF	-INF	+INF	if z < 0

x	y	z	r=(x*y) +z	r=(x*y) - z	r = -(x*y)+z	r= -(x*y)-z	Comment
0	0	0	0	0	0	0	The sign of the result depends on the sign of the operands and on the rounding mode. The product x*y is +0 or -0, depending on the signs of x and y. The summation/subtraction of the zero representing (x*y) and the zero representing z can lead to one of the four cases shown in Table 2-1.
0	F	0	0	0	0	0	
F	0	0	0	0	0	0	
0	0	F	z	-z	z	-z	
0	F	F	z	-z	z	-z	
F	0	F	z	-z	z	-z	
F	F	0	x*y	x*y	-x*y	-x*y	Rounded to the destination precision, with bounded exponent
F	F	F	(x*y)+z	(x*y)-z	-(x*y)+z	-(x*y)-z	Rounded to the destination precision, with bounded exponent; however, if the exact values of x*y and z are equal in magnitude with signs resulting in the FMA operation producing 0, the rounding behavior described in Table 2-1.

If unmasked floating-point exceptions are signaled (invalid operation, denormal operand, overflow, underflow, or inexact result) the result register is left unchanged and a floating-point exception handler is invoked.

2.3.1 FMA Instruction Operand Order and Arithmetic Behavior

FMA instruction mnemonics are defined explicitly with an ordered three digits, e.g. VFMADD132PD. The value of each digit refer to the ordering of the three source operand as defined by instruction encoding specification:

- 1: The first source operand (also the destination operand) in the syntactical order listed in this specification.
- 2: The second source operand in the syntactical order. This is a YMM/XMM register, encoded using VEX prefix.
- 3: The third source operand in the syntactical order. The first and third operand are encoded following ModR/M encoding rules.

The ordering of each digit within the mnemonic refers to the floating-point data listed on the right-hand side of the arithmetic equation of each FMA operation (see Table 2-2):

- The first position in the three digit ordering of a FMA mnemonic refers to the first FP data expressed in the arithmetic equation of FMA operation, the multiplicand.
- The second position in the three digit FMA mnemonic refers to the second FP data expressed in the arithmetic equation of FMA operation, the multiplier.
- The third position in the three digit FMA mnemonic refers to the FP data being added/subtracted to the multiplication result.

Note non-numerical result of an FMA operation do not resemble the mathematically-defined commutative property between the multiplicand and the multiplier values (see Table 2-2). Consequently, software tools (such as an assembler) may support a complementary set of FMA mnemonics for each FMA instruction for ease of programming to take advantage of the mathematical property of commutative multiplications. For example, an assembler may optionally support the complementary mnemonic "VFMADD312PD" in addition to the true mnemonic "VFMADD132PD". The assembler will generate the same instruction opcode sequence corresponding to VFMADD132PD. The processor executes VFMADD132PD and report any NAN conditions based on the definition of VFMADD132PD. Similarly, if the complementary mnemonic VFMADD123PD is supported by an assembler at source level, it must generate the opcode sequence corresponding to VFMADD213PD; the complementary mnemonic VFMADD321PD must produce the opcode sequence defined by VFMADD231PD. In the absence of FMA operations reporting a NAN result, the numerical results of using either mnemonic with an assembler supporting both mnemonics will match the behavior defined in Table 2-2. Support for the complementary FMA mnemonics by software tools is optional.

2.4 ACCESSING YMM REGISTERS

The lower 128 bits of a YMM register is aliased to the corresponding XMM register. Legacy SSE instructions (i.e. SIMD instructions operating on XMM state but not using the VEX prefix, also referred to non-VEX encoded SIMD instructions) will not access

the upper bits (255:128) of the YMM registers. AVX and FMA instructions with a VEX prefix and vector length of 128-bits zeroes the upper 128 bits of the YMM register. See Chapter 2, "Programming Considerations with 128-bit SIMD Instructions" for more details.

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

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

2.5 MEMORY ALIGNMENT

Memory alignment requirements on VEX-encoded instruction differs from non-VEX-encoded instructions. Memory alignment applies to non-VEX-encoded SIMD instructions in three categories:

- Explicitly-aligned SIMD load and store instructions accessing 16 bytes of memory (e.g. MOVAPD, MOVAPS, MOVDQA, etc.). These instructions always require memory address to be aligned on 16-byte boundary.
- Explicitly-unaligned SIMD load and store instructions accessing 16 bytes or less of data from memory (e.g. MOVUPD, MOVUPS, MOVDQU, MOVQ, MOVD, etc.). These instructions do not require memory address to be aligned on 16-byte boundary.
- The vast majority of arithmetic and data processing instructions in legacy SSE instructions (non-VEX-encoded SIMD instructions) support memory access semantics. When these instructions access 16 bytes of data from memory, the memory address must be aligned on 16-byte boundary.

Most arithmetic and data processing instructions encoded using the VEX prefix and performing memory accesses have more flexible memory alignment requirements than instructions that are encoded without the VEX prefix. Specifically,

- With the exception of explicitly aligned 16 or 32 byte SIMD load/store instructions, most VEX-encoded, arithmetic and data processing instructions operate in a flexible environment regarding memory address alignment, i.e. VEX-encoded instruction with 32-byte or 16-byte load semantics will support unaligned load operation by default. Memory arguments for most instructions with VEX prefix operate normally without causing #GP(0) on any byte-granularity alignment (unlike Legacy SSE instructions). The instructions that require explicit memory alignment requirements are listed in Table 2-4.

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

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

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

Certain AVX instructions always require 16- or 32-byte alignment (see the complete list of such instructions in Table 2-4). These instructions will #GP(0) if not aligned to 16-byte boundaries (for 16-byte granularity loads and stores) or 32-byte boundaries (for 32-byte loads and stores).

Table 2-3. Alignment Faulting Conditions when Memory Access is Not Aligned

EFLAGS.AC==1 && Ring-3 && CR0.AM == 1			0	1
Instruction Type	AVX, FMA,	16- or 32-byte "explicitly unaligned" loads and stores (see Table 2-5)	no fault	no fault
		VEX op YMM, m256	no fault	no fault
		VEX op XMM, m128	no fault	no fault
		"explicitly aligned" loads and stores (see Table 2-4)	#GP(0)	#GP(0)
		2, 4, or 8-byte loads and stores	no fault	#AC(0)
	SSE	16 byte "explicitly unaligned" loads and stores (see Table 2-5)	no fault	no fault
		op XMM, m128	#GP(0)	#GP(0)
		"explicitly aligned" loads and stores (see Table 2-4)	#GP(0)	#GP(0)
		2, 4, or 8-byte loads and stores	no fault	#AC(0)

Table 2-4. Instructions Requiring Explicitly Aligned Memory

Require 16-byte alignment	Require 32-byte alignment
(V)MOVDQA xmm, m128	VMOVDQA ymm, m256
(V)MOVDQA m128, xmm	VMOVDQA m256, ymm
(V)MOVAPS xmm, m128	VMOVAPS ymm, m256
(V)MOVAPS m128, xmm	VMOVAPS m256, ymm
(V)MOVAPD xmm, m128	VMOVAPD ymm, m256

Table 2-4. Instructions Requiring Explicitly Aligned Memory

Require 16-byte alignment	Require 32-byte alignment
(V)MOVAPD m128, xmm	VMOVAPD m256, ymm
(V)MOVNTPS m128, xmm	VMOVNTPS m256, ymm
(V)MOVNTPD m128, xmm	VMOVNTPD m256, ymm
(V)MOVNTDQ m128, xmm	VMOVNTDQ m256, ymm
(V)MOVNTDQA xmm, m128	

Table 2-5. Instructions Not Requiring Explicit Memory Alignment

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

2.6 SIMD FLOATING-POINT EXCEPTIONS

AVX and FMA instructions can generate SIMD floating-point exceptions (#XM) and respond to exception masks in the same way as Legacy SSE instructions. When CR4.OSXMMEXCPT=0 any unmasked FP exceptions generate an Undefined Opcode exception (#UD).

AVX FP exceptions are created in a similar fashion (differing only in number of elements) to Legacy SSE and SSE2 instructions capable of generating SIMD floating-point exceptions.

AVX introduces no new arithmetic operations (AVX floating-point are analogues of existing Legacy SSE instructions). FMA introduces new arithmetic operations, detailed FMA numeric behavior are described in Section 2.3.

2.7 INSTRUCTION EXCEPTION SPECIFICATION

To use this reference of instruction exceptions, look at each instruction for a description of the particular exception type of interest. For example, ADDPS contains the entry:

"See Exceptions Type 2"

In this entry, *"Type2"* can be looked up in Table 2-6.

The instruction's corresponding CPUID feature flag can be identified in the fourth column of the Instruction summary table.

Note: #UD on CPUID feature flags=0 is not guaranteed in a virtualized environment if the hardware supports the feature flag.

Table 2-6. Exception class description

Exception Class	Instruction set	Mem arg	Floating-Point Exceptions (#XM)
Type 1	AVX, Legacy SSE	16/32 byte explicitly aligned	none
Type 2	AVX, FMA Legacy SSE	16/32 byte not explicitly aligned	yes
Type 3	AVX, FMA Legacy SSE	< 16 byte	yes
Type 4	AVX, Legacy SSE	16/32 byte not explicitly aligned	no
Type 5	AVX, Legacy SSE	< 16 byte	no
Type 6	AVX (no Legacy SSE)	Varies	(At present, none do)
Type 7	AVX, Legacy SSE	none	none
Type 8	AVX	none	none
Type 9	AVX	4 byte	none
Type 11	F16C	8 or 16 byte	yes

See Table 2-7 for lists of instructions in each exception class.

Table 2-7. Instructions in each Exception Class

Exception Class	Instruction
Type 1	(V)MOVAPD, (V)MOVAPS, (V)MOVDQA, (V)MOVNTDQ, (V)MOVNTDQA, (V)MOVNTPD, (V)MOVNTPS
Type 2	(V)ADDPD, (V)ADDPS, (V)ADDSUBPD, (V)ADDSUBPS, (V)CMPPD, (V)CMPPS, (V)CVTDQ2PS, (V)CVTPD2DQ, (V)CVTPD2PS, (V)CVTPS2DQ, (V)CVTTPD2DQ, (V)CVTTPS2DQ, (V)DIVPD, (V)DIVPS, (V)DPPD*, (V)DPPS*, VFMADD132PD, VFMADD213PD, VFMADD231PD, VFMADD132PS, VFMADD213PS, VFMADD231PS, VFMADDSUB132PD, VFMADDSUB213PD, VFMADDSUB231PD, VFMADDSUB132PS, VFMADDSUB213PS, VFMADDSUB231PS, VFMSUBADD132PD, VFMSUBADD213PD, VFMSUBADD231PD, VFMSUBADD132PS, VFMSUBADD213PS, VFMSUBADD231PS, VFMSUB132PD, VFMSUB213PD, VFMSUB231PD, VFMSUB132PS, VFMSUB213PS, VFMSUB231PS, VFNMADD132PD, VFNMADD213PD, VFNMADD231PD, VFNMADD132PS, VFNMADD213PS, VFNMADD231PS, VFNMSUB132PD, VFNMSUB213PD, VFNMSUB231PD, VFNMSUB132PS, VFNMSUB213PS, VFNMSUB231PS, (V)HADDPD, (V)HADDPS, (V)HSUBPD, (V)HSUBPS, (V)MAXPD, (V)MAXPS, (V)MINPD, (V)MINPS, (V)MULPD, (V)MULPS, (V)ROUNDPS, (V)ROUNDPS, (V)SQRTPD, (V)SQRTPS, (V)SUBPD, (V)SUBPS
Type 3	(V)ADDSD, (V)ADDSS, (V)CMPSD, (V)CMPSS, (V)COMISD, (V)COMISS, (V)CVTPS2PD, (V)CVTSD2SI, (V)CVTSD2SS, (V)CVTSI2SD, (V)CVTSI2SS, (V)CVTSS2SD, (V)CVTSS2SI, (V)CVTTSD2SI, (V)CVTTSS2SI, (V)DIVSD, (V)DIVSS, VFMADD132SD, VFMADD213SD, VFMADD231SD, VFMADD132SS, VFMADD213SS, VFMADD231SS, VFMSUB132SD, VFMSUB213SD, VFMSUB231SD, VFMSUB132SS, VFMSUB213SS, VFMSUB231SS, VFNMADD132SD, VFNMADD213SD, VFNMADD231SD, VFNMADD132SS, VFNMADD213SS, VFNMADD231SS, VFNMSUB132SD, VFNMSUB213SD, VFNMSUB231SD, VFNMSUB132SS, VFNMSUB213SS, VFNMSUB231SS, (V)MAXSD, (V)MAXSS, (V)MINSD, (V)MINSS, (V)MULSD, (V)MULSS, (V)ROUNDSD, (V)ROUNDSS, (V)SQRTSD, (V)SQRTSS, (V)SUBSD, (V)SUBSS, (V)UCOMISD, (V)UCOMISS

Exception Class	Instruction
Type 4	(V)AESDEC, (V)AESDECLAST, (V)AESENC, (V)AESENCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDPS, (V)ANDNPD, (V)ANDNPS, (V)BLENDPD, (V)BLENDPS, VBLENDVPD, VBLENDVPS, (V)LDDQU, (V)MASKMOVDQU, (V)PTEST, VTESTPS, VTESTPD, (V)MOVDQU*, (V)MOVSHDUP, (V)MOVSLDUP, (V)MOVUPD*, (V)MOVUPS*, (V)MPSADBW, (V)ORPD, (V)ORPS, (V)PABSB, (V)PABSW, (V)PABSD, (V)PACKSSWB, (V)PACKSSDW, (V)PACKUSWB, (V)PACKUSDW, (V)PADDB, (V)PADDW, (V)PADDD, (V)PADDQ, (V)PADDSB, (V)PADDSW, (V)PADDUSB, (V)PADDUSW, (V)PALIGNR, (V)PAND, (V)PANDN, (V)PAVGB, (V)PAVGW, (V)PBLENDVB, (V)PBLENDW, (V)PCMP(E/I)STRI/M, (V)PCMPEQB, (V)PCMPEQW, (V)PCMPEQD, (V)PCMPEQQ, (V)PCMPGTB, (V)PCMPGTW, (V)PCMPGTD, (V)PCMPGTQ, (V)PCLMULQDQ, (V)PHADDW, (V)PHADDD, (V)PHADDSW, (V)PHMINPOSUW, (V)PHSUBD, (V)PHSUBW, (V)PHSUBSW,
	(V)PMADDWD, (V)PMADDUBSW, (V)PMAXSB, (V)PMAXSW, (V)PMAXSD, (V)PMAXUB, (V)PMAXUW, (V)PMAXUD, (V)PMINSB, (V)PMINSW, (V)PMINSD, (V)PMINUB, (V)PMINUW, (V)PMINUD, (V)PMULHUW, (V)PMULHRSW, (V)PMULHW, (V)PMULLW, (V)PMULLD, (V)PMULUDQ, (V)PMULDQ, (V)POR, (V)PSADBW, (V)PSHUFB, (V)PSHUFD, (V)PSHUFHW, (V)PSHUFLW, (V)PSIGNB, (V)PSIGNW, (V)PSIGND, (V)PSLLW, (V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ, (V)PSUBB, (V)PSUBW, (V)PSUBD, (V)PSUBQ, (V)PSUBSB, (V)PSUBSW, (V)PUNPCKHBW, (V)PUNPCKHWD, (V)PUNPCKHDQ, (V)PUNPCKHQDQ, (V)PUNPCKLBW, (V)PUNPCKLWD, (V)PUNPCKLDQ, (V)PUNPCKLQDQ, (V)PXOR, (V)RCPPS, (V)RSQRTPS, (V)SHUFPD, (V)SHUFPS, (V)UNPCKHPD, (V)UNPCKHPS, (V)UNPCKLPD, (V)UNPCKLPS, (V)XORPD, (V)XORPS
Type 5	(V)CVTDQ2PD, (V)EXTRACTPS, (V)INSERTPS, (V)MOVD, (V)MOVQ, (V)MOVDDUP, (V)MOVLPD, (V)MOVLPS, (V)MOVHPD, (V)MOVHPS, (V)MOVSD, (V)MOVSS, (V)PEXTRB, (V)PEXTRD, (V)PEXTRW, (V)PEXTRQ, (V)PINSRB, (V)PINSRD, (V)PINSRW, (V)PINSRQ, (V)RCPSS, (V)RSQRTSS, (V)PMOVSX/ZX
Type 6	VEXTRACTF128, VPERMILPD, VPERMILPS, VPERM2F128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS**, VMASKMOVPD**
Type 7	(V)MOVLHPS, (V)MOVHLPS, (V)MOVMSKPD, (V)MOVMSKPS, (V)PMOVMSKB, (V)PSLLDQ, (V)PSRLDQ, (V)PSLLW, (V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ
Type 8	VZEROALL, VZEROUPPER
Type 9	VLDMXCSR*, VSTMXCSR
Type 11	VCVTPH2PS, VCVTPS2PH

(*) - Additional exception restrictions are present - see the Instruction description for details

(**) - Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s, i.e. no alignment checks are performed.

Table 2-7 classifies exception behaviors for AVX instructions. Within each class of exception conditions that are listed in Table 2-10 through Table 2-16, certain subsets of AVX instructions may be subject to #UD exception depending on the encoded value of the VEX.L field. Table 2-9 provides supplemental information of AVX instructions that may be subject to #UD exception if encoded with incorrect values in the VEX.W or VEX.L field.

Table 2-8. #UD Exception and VEX.W=1 Encoding

Exception Class	#UD If VEX.W = 1 in all modes	#UD If VEX.W = 1 in non-64-bit modes
Type 1		
Type 2		
Type 3		
Type 4	VBLENDVPD, VBLENDVPS, VPBLENDVB, VTESTPD, VTESTPS	
Type 5		VPEXTRQ, VPINSRQ,
Type 6	VEXTRACTF128, VPERMILPD, VPERMILPS, VPERM2F128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS, VMASKMOVPD	
Type 7		
Type 8	VZEROALL, VZEROUPPER	
Type 9		
Type 11	VCVTPH2PS, VCVTPS2PH	

Table 2-9. #UD Exception and VEX.L Field Encoding

Exception Class	#UD If VEX.L = 0	#UD If VEX.L = 1
Type 1		VMOVNTDQA
Type 2		VDPPD
Type 3		
Type 4		VMASKMOVDQU, VMPSADBW, VPABSB/W/D, VPACKSSWB/DW, VPACKUSWB/DW, VPADDB/W/D, VPADDQ, VPADDSB/W, VPADDUSB/W, VPALIGNR, VPAND, VPANDN, VPAVGB/W, VPBLENDVB, VPBLENDW, VPCMP(E/I)STRI/M, VPCMPEQB/W/D/Q, VPCMPGTB/W/D/Q, VPHADDW/D, VPHADDSW, VPHMINPOSUW, VPHSUBD/W, VPHSUBSW, VPMADDWD, VPMADDUBSW, VPMAXSB/W/D, VPMAXUB/W/D, VPMINSB/W/D, VPMINUB/W/D, VPMULHUW, VPMULHRSW, VPMULHW/LW, VPMULLD, VPMULUDQ, VPMULDQ, VPOR, VPSADBW, VPSHUFB/D, VPSHUFHW/LW, VPSIGNB/W/D, VPSLLW/D/Q, VPSRAW/D, VPSRLW/D/Q, VPSUBB/W/D/Q, VPSUBSB/W, VPUNPCKHBW/WD/DQ, VPUNPCKHQDQ, VPUNPCKLBW/WD/DQ, VPUNPCKLQDQ, VPXOR
Type 5		VEXTRACTPS, VINSERTPS, VMOVD, VMOVQ, VMOVLPD, VMOVLPS, VMOVHPD, VMOVHPS, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ, VPMOVSX/ZX
Type 6	VEXTRACTF128, VPERM2F128, VBROADCASTSD, VBROADCASTF128, VINSERTF128,	
Type 7		VMOVLHPS, VMOVHLPS, VPMOVMSKB, VPSLLDQ, VPSRLDQ, VPSLLW, VPSLLD, VPSLLQ, VPSRAW, VPSRAD, VPSRLW, VPSRLD, VPSRLQ
Type 8		
Type 9		VLDMXCSR, VSTMXCSR

2.7.1 Exceptions Type 1 (Aligned memory reference)

Table 2-10. Type 1 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix
			X	X	VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X	X	VEX.256: Memory operand is not 32-byte aligned VEX.128: Memory operand is not 16-byte aligned
	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault

2.7.2 Exceptions Type 2 (>=16 Byte Memory Reference, Unaligned)

Table 2-11. Type 2 Class Exception Conditions

Exception	Real	Virtual 8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1

2.7.3 Exceptions Type 3 (<16 Byte memory argument)

Table 2-12. Type 3 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1

2.7.4 Exceptions Type 4 (>=16 Byte mem arg no alignment, no floating-point exceptions)

Table 2-13. Type 4 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix
			X	X	VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault

2.7.5 Exceptions Type 5 (<16 Byte mem arg and no FP exceptions)

Table 2-14. Type 5 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix
			X	X	VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

2.7.6 Exceptions Type 6 (VEX-Encoded Instructions Without Legacy SSE Analogues)

Note: At present, the AVX instructions in this category do not generate floating-point exceptions.

Table 2-15. Type 6 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix
			X	X	If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
			X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
			X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	For a page fault
Alignment Check #AC(0)			X	X	For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

2.7.7 Exceptions Type 7 (No FP exceptions, no memory arg)

Table 2-16. Type 7 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix
			X	X	VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1

2.7.8 Exceptions Type 8 (AVX and no memory argument)

Table 2-17. Type 8 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			Always in Real or Virtual 80x86 mode
			X	X	If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0. If CPUID.01H.ECX.AVX[bit 28]=0. If VEX.vvvv != 1111B.
	X	X	X	X	If proceeded by a LOCK prefix (F0H)
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.

2.7.9 Exception Type 9 (AVX)

Table 2-18. Type 9 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			Always in Real or Virtual 80x86 mode
			X	X	If CR4.OSXSAVE[bit 18]=0. If CR0.EM[bit 2] = 1. If CPUID.01H.ECX.AVX[bit 28]=0 If VEX.L = 1
			X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	For a page fault
Alignment Check #AC(0)			X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

2.7.10 Exception Type 11 (VEX-only, mem arg no AC, floating-point exceptions)

Table 2-19. Type 11 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix
			X	X	VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1

2.8 PROGRAMMING CONSIDERATIONS WITH 128-BIT SIMD INSTRUCTIONS

VEX-encoded SIMD instructions generally operate on the 256-bit YMM register state. In contrast, non-VEX encoded instructions (e.g from SSE to AES) operating on XMM registers only access the lower 128-bit of YMM registers. Processors supporting both

256-bit VEX-encoded instruction and legacy 128-bit SIMD instructions has internal state to manage the upper and lower halves of the YMM states. Functionally, VEX-encoded SIMD instructions can be intermixed with legacy SSE instructions (non-VEX-encoded SIMD instructions operating on XMM registers). However, there is a performance impact with intermixing VEX-encoded SIMD instructions (AVX, FMA) and Legacy SSE instructions that only operate on the XMM register state.

The general programming considerations to realize optimal performance are the following:

- Minimize transition delays and partial register stalls with YMM registers accesses: Intermixed 256-bit, 128-bit or scalar SIMD instructions that are encoded with VEX prefixes have no transition delay due to internal state management.

  Sequences of legacy SSE instructions (including SSE2, and subsequent generations non-VEX-encoded SIMD extensions) that are not intermixed with VEX-encoded SIMD instructions are not subject to transition delays.
- When an application must employ AVX and/or FMA, along with legacy SSE code, it should minimize the number of transitions between VEX-encoded instructions and legacy, non-VEX-encoded SSE code. Section 2.8.1 provides recommendation for software to minimize the impact of transitions between VEX-encoded code and legacy SSE code.

2.8.1 Clearing Upper YMM State Between AVX and Legacy SSE Instructions

There is no transition penalty if an application clears the upper bits of all YMM registers (set to '0') via VZEROUPPER, VZEROALL, before transitioning between AVX instructions and legacy SSE instructions. Note: clearing the upper state via sequences of XORPS or loading '0' values individually may be useful for breaking dependency, but will not avoid state transition penalties.

Example 1: an application using 256-bit AVX instructions makes calls to a library written using Legacy SSE instructions. This would encounter a delay upon executing the first Legacy SSE instruction in that library and then (after exiting the library) upon executing the first AVX instruction. To eliminate both of these delays, the user should execute the instruction VZEROUPPER prior to entering the legacy library and (after exiting the library) before executing in a 256-bit AVX code path.

Example 2: a library using 256-bit AVX instructions is intended to support other applications that uses legacy SSE instructions. Such a library function should execute VZEROUPPER prior to executing other VEX-encoded instructions. The library function should issue VZEROUPPER at the end of the function before it returns to the calling application. This will prevent the calling application to experience delay when it starts to execute legacy SSE code.

2.8.2 Using AVX 128-bit Instructions Instead of Legacy SSE instructions

Applications using AVX and FMA should migrate legacy 128-bit SIMD instructions to their 128-bit AVX equivalents. AVX supplies the full complement of 128-bit SIMD instructions except for AES and PCLMULQDQ.

2.8.3 Unaligned Memory Access and Buffer Size Management

The majority of AVX instructions support loading 16/32 bytes from memory without alignment restrictions (A number non-VEX-encoded SIMD instructions also don't require 16-byte address alignment, e.g. MOVDQU, MOVUPS, MOVUPD, LDDQU, PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM). A buffer size management issue related to unaligned SIMD memory access is discussed here.

The size requirements for memory buffer allocation should consider unaligned SIMD memory semantics and application usage. Frequently a caller function may pass an address pointer in conjunction with a length parameter. From the caller perspective, the length parameter usually corresponds to the limit of the allocated memory buffer range, or it may corresponds to certain application-specific configuration parameter that have indirect relationship with valid buffer size.

For certain types of application usage, it may be desirable to make distinctions between valid buffer range limit versus other application specific parameters related memory access patterns, examples of the latter may be stride distance, frame dimensions, etc. There may be situations that a callee wishes to load 16-bytes of data with parts of the 16-bytes lying outside the valid memory buffer region to take advantage of the efficiency of SIMD load bandwidth and discard invalid data elements outside the buffer boundary. An example of this may be in video processing of frames having dimensions that are not modular 16 bytes.

To support the added margin of safety in situations of buffer size allocation and iterative pointer advancement occurring across modules of different software visibility. The standard programming practice of caller function allocation of buffer size based on non-SIMD processing requirement should consider an added padding size to support newer SIMD extensions offering more lax alignment restrictions. The extra padding space can prevent the rare occurrence of access rights violation described below:

- A present page in the linear address space being used by ring 3 code is followed by a page owned by ring 0 code,
- A caller routine allocates a memory buffer without adding extra pad space and passes the buffer address to a callee routine,
- A callee routine implements an iterative processing algorithm by advancing an address pointer relative to the buffer address using SIMD instructions with unaligned 16/32 load semantics
- The callee routine may choose to load 16/32 bytes near buffer boundary with the intent to discard invalid data outside the data buffer allocated by the caller.

- If the valid data buffer extends to the end of the present page, unaligned 16/32 byte loads near the end of a present page may spill over to the subsequent ring-0 page and causing a #GP.

As a general rule, the minimal padding size should be the width the SIMD register that might be used in conjunction with unaligned SIMD memory access.

2.9 CPUID INSTRUCTION

CPUID—CPU Identification

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

Description

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

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

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

Table 2-20 shows information returned, depending on the initial value loaded into the EAX register. Table 2-21 shows the maximum CPUID input value recognized for each family of IA-32 processors on which CPUID is implemented.

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

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

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

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

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

See also:

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

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

Table 2-20. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
0H	*Basic CPUID Information*	
	EAX	Maximum Input Value for Basic CPUID Information (see Table 2-21)
	EBX	"Genu"
	ECX	"ntel"
	EDX	"inel"
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 2-2)
	EBX	Bits 7-0: Brand Index Bits 15-8: CLFLUSH line size (Value ∗ 8 = cache line size in bytes) Bits 23-16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31-24: Initial APIC ID
	ECX	Feature Information (see Figure 2-3 and Table 2-23)
	EDX	Feature Information (see Figure 2-4 and Table 2-24)
		NOTES: * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the maximum number of unique initial APIC IDs reserved for addressing different logical processors in a physical package.

Table 2-20. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor
02H	EAX Cache and TLB Information (see Table 2-25) EBX Cache and TLB Information ECX Cache and TLB Information EDX Cache and TLB Information
03H	Reserved. EAX Reserved. EBX Bits 00-31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) ECX EDX Bits 32-63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
	NOTES: Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature.
	See AP-485, *Intel Processor Identification and the CPUID Instruction* (Order Number 241618) for more information on PSN.
	CPUID leaves > 3 < 80000000 are visible only when IA32_MISC_ENABLES.BOOT_NT4[bit 22] = 0 (default).
	Deterministic Cache Parameters Leaf
04H	**NOTES:** Leaf 04H output depends on the initial value in ECX. See also: "INPUT EAX = 4: Returns Deterministic Cache Parameters for each level on page 2-56. EAX Bits 4-0: Cache Type Field 0 = Null - No more caches 1 = Data Cache 2 = Instruction Cache 3 = Unified Cache 4-31 = Reserved Bits 7-5: Cache Level (starts at 1) Bits 8: Self Initializing cache level (does not need SW initialization) Bits 9: Fully Associative cache

Table 2-20. Information Returned by CPUID Instruction(Continued)

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

Table 2-20. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bits 15-00: Largest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0
	ECX	Bits 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported Bits 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled Bits 31 - 02: Reserved
	EDX	Bits 03 - 00: Number of C0* sub C-states supported using MWait Bits 07 - 04: Number of C1* sub C-states supported using MWAIT Bits 11 - 08: Number of C2* sub C-states supported using MWAIT Bits 15 - 12: Number of C3* sub C-states supported using MWAIT Bits 19 - 16: Number of C4* sub C-states supported using MWAIT Bits 31 - 20: Reserved = 0 **NOTE:** * The definition of C0 through C4 states for MWAIT extension are processor-specific C-states, not ACPI C-states.
	Thermal and Power Management Leaf	
6H	EAX	Bits 00: Digital temperature sensor is supported if set Bits 01: Intel Turbo Boost Technology is available Bits 31 - 02: Reserved
	EBX	Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor Bits 31 - 04: Reserved
	ECX	Bits 00: Hardware Coordination Feedback Capability (Presence of MCNT and ACNT MSRs). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of expected processor performance at frequency specified in CPUID Brand String Bits 02 - 01: Reserved = 0 Bit 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H) Bits 31 - 04: Reserved = 0
	EDX	Reserved = 0
	Structured Extended feature Leaf	
07H		**NOTES:** Leaf 07H main leaf (ECX = 0). IF leaf 07H is not supported, EAX=EBX=ECX=EDX=0

Table 2-20. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EAX	Bits 31-0: Reports the maximum number sub-leaves that are supported in leaf 07H.
	EBX	Bits 00: FGSBASE Bits 31-1: Reserved.
	ECX	Bit 31-0: Reserved
	EDX	Bit 31-0: Reserved.
	Structured Extended Feature Enumeration Sub-leaves (EAX = 07H, ECX = n, n > 1)	
07H		**NOTES:** Leaf 07H output depends on the initial value in ECX. If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0.
	EAX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	EBX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	ECX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	EDX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	Direct Cache Access Information Leaf	
09H	EAX	Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H)
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
	Architectural Performance Monitoring Leaf	
0AH	EAX	Bits 07 - 00: Version ID of architectural performance monitoring Bits 15- 08: Number of general-purpose performance monitoring counter per logical processor Bits 23 - 16: Bit width of general-purpose, performance monitoring counter Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events

Table 2-20. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bit 0: Core cycle event not available if 1 Bit 1: Instruction retired event not available if 1 Bit 2: Reference cycles event not available if 1 Bit 3: Last-level cache reference event not available if 1 Bit 4: Last-level cache misses event not available if 1 Bit 5: Branch instruction retired event not available if 1 Bit 6: Branch mispredict retired event not available if 1 Bits 31- 07: Reserved = 0
	ECX EDX	Reserved = 0 Bits 04 - 00: Number of fixed-function performance counters (if Version ID > 1) Bits 12- 05: Bit width of fixed-function performance counters (if Version ID > 1) Reserved = 0
	Extended Topology Enumeration Leaf	
0BH	EAX	**NOTES:** Most of Leaf 0BH output depends on the initial value in ECX. EDX output do not vary with initial value in ECX. ECX[7:0] output always reflect initial value in ECX. All other output value for an invalid initial value in ECX are 0 This leaf exists if EBX[15:0] contain a non-zero value. Bits 4-0: Number of bits to shift right on x2APIC ID to get a unique topology ID of the next level type*. All logical processors with the same next level ID share current level. Bits 31-5: Reserved.
0BH	EBX	Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**. Bits 31- 16: Reserved.
	ECX	Bits 07 - 00: Level number. Same value in ECX input Bits 15 - 08: Level type***. Bits 31 - 16: Reserved.
	EDX	Bits 31- 0: x2APIC ID the current logical processor.

Table 2-20. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor
	NOTES: * Software should use this field (EAX[4:0]) to enumerate processor topology of the system.
	** Software must not use EBX[15:0] to enumerate processor topology of the system. This value in this field (EBX[15:0]) is only intended for display/diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations. *** The value of the "level type" field is not related to level numbers in any way, higher "level type" values do not mean higher levels. Level type field has the following encoding: 0: invalid 1: SMT 2: Core 3-255: Reserved
	Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)
0DH	**NOTES:** Leaf 0DH main leaf (ECX = 0). EAX: Bits 31-0: Reports the valid bit fields of the lower 32 bits of the XFEATURE_ENABLED_MASK register. If a bit is 0, the corresponding bit field in XFEATURE_ENABLED_MASK is reserved. Bit 00: legacy x87 Bit 01: 128-bit SSE Bit 02: 256-bit AVX EBX: Bits 31-0: Maximum size (bytes) required by enabled features in XFEATURE_ENABLED_MASK. May be different than ECX when features at the end of the save area are not enabled. ECX: Bit 31-0: Maximum size (bytes) of the XSAVE/XRSTOR save area required by all HW supported features, i.e all the valid bit fields in XFEATURE_ENABLED_MASK. This includes the size needed for the XSAVE.HEADER.

Table 2-20. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bit 31-0: Reports the valid bit fields of the upper 32 bits of the XFEATURE_ENABLED_MASK register. If a bit is 0, the corresponding bit field in XFEATURE_ENABLED_MASK is reserved
	Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)	
	EAX	Bit 00: XSAVEOPT is available; Bits 31-1: Reserved
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
	Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n > 1)	
0DH		**NOTES:** Leaf 0DH output depends on the initial value in ECX. If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0.
	EAX	Bits 31-0: The size in bytes of the save area for an extended state associated with a valid sub-leaf index, *n*. Each valid sub-leaf index maps to a valid bit in XFEATURE_ENABLED_MASK starting at bit position 2. This field reports 0 if the sub-leaf index, *n*, is invalid*.
	EBX	Bits 31-0: The offset in bytes of the save area from the beginning of the XSAVE/XRSTOR area. This field reports 0 if the sub-leaf index, *n*, is invalid*.
	ECX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	EDX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
		*The highest valid sub-leaf index, *n*, is (POPCNT(CPUID.(EAX=0D, ECX=0):EAX) + POPCNT(CPUID.(EAX=0D, ECX=0):EDX) - 1)
	Extended Function CPUID Information	
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information (see Table 2-21).
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved

Table 2-20. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
80000001H	EAX	Extended Processor Signature and Feature Bits.
	EBX	Reserved
	ECX	Bit 0: LAHF/SAHF available in 64-bit mode Bits 31-1 Reserved
	EDX	Bits 10-0: Reserved Bit 11: SYSCALL/SYSRET available (when in 64-bit mode) Bits 19-12: Reserved = 0 Bit 20: Execute Disable Bit available Bits 28-21: Reserved = 0 Bit 29: Intel® 64 Architecture available if 1 Bits 31-30: Reserved = 0
80000002H	EAX EBX ECX EDX	Processor Brand String Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued
80000003H	EAX EBX ECX EDX	Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued
80000004H	EAX EBX ECX EDX	Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued
80000005H	EAX EBX ECX EDX	Reserved = 0 Reserved = 0 Reserved = 0 Reserved = 0
80000006H	EAX EBX	Reserved = 0 Reserved = 0
	ECX	Bits 7-0: Cache Line size in bytes Bits 15-12: L2 Associativity field * Bits 31-16: Cache size in 1K units
	EDX	Reserved = 0

Table 2-20. Information Returned by CPUID Instruction(Continued)

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

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

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register (see Table 2-21) and is processor specific.

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

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

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

Information

When CPUID executes with EAX set to 0, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register (see Table 2-21) and is processor specific.

Table 2-21. Highest CPUID Source Operand for Intel 64 and IA-32 Processors

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

IA32_BIOS_SIGN_ID Returns Microcode Update Signature

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

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

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

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

See Table 2-22 for available processor type values. Stepping IDs are provided as needed.

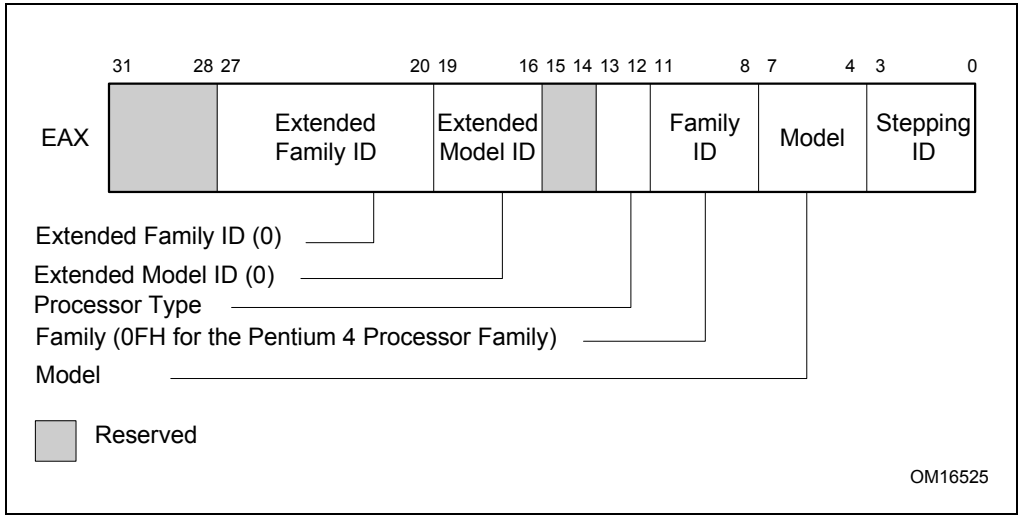


Figure 2-2. Version Information Returned by CPUID in EAX

Table 2-22. Processor Type Field

Type	Encoding
Original OEM Processor	00B
Intel OverDrive® Processor	01B

Table 2-22. Processor Type Field

Type	Encoding
Dual processor (not applicable to Intel486 processors)	10B
Intel reserved	11B

NOTE

See "Caching Translation Information" in Chapter 4, "Paging," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, and Chapter 14 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for information on identifying earlier IA-32 processors.

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

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

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

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

INPUT EAX = 1: Returns Additional Information in EBX

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

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

- Local APIC ID (high byte of EBX) — this number is the 8-bit ID that is assigned to the local APIC on the processor during power up. This field was introduced in the Pentium 4 processor.

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

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

- Figure 2-3 and Table 2-23 show encodings for ECX.
- Figure 2-4 and Table 2-24 show encodings for EDX.

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

NOTE

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

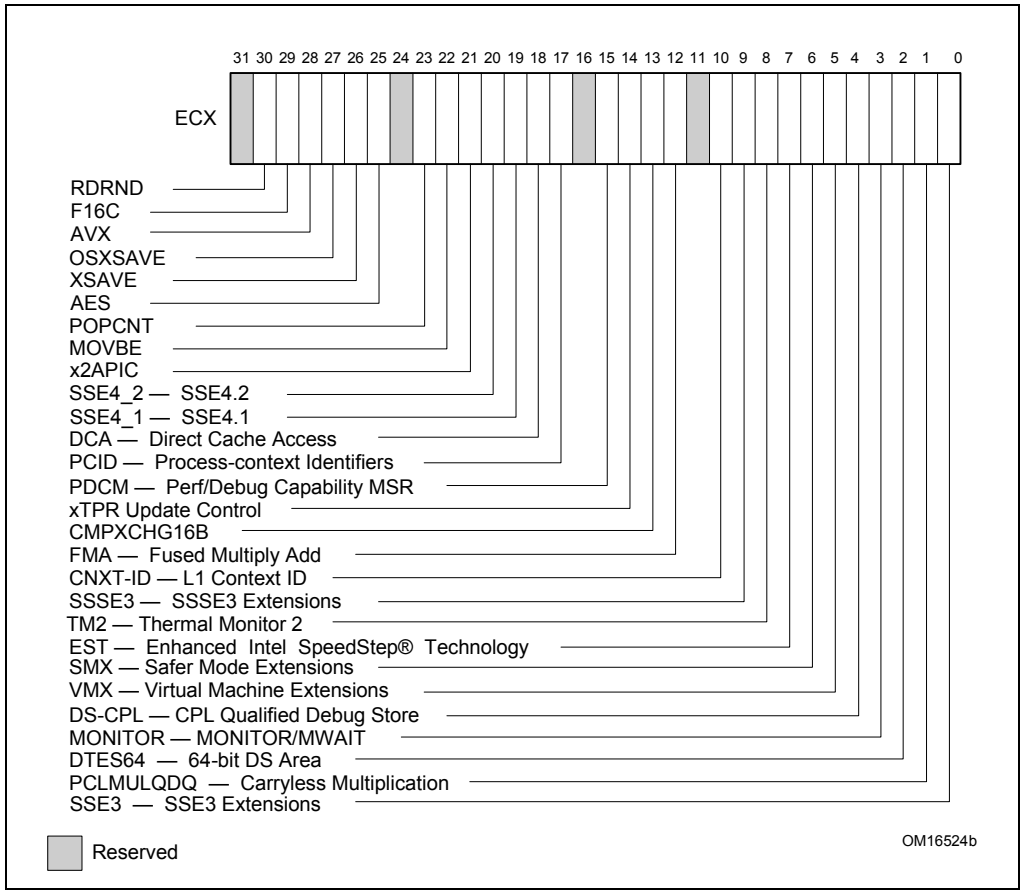


Figure 2-3. Feature Information Returned in the ECX Register

Table 2-23. Feature Information Returned in the ECX Register

Bit #	Mnemonic	Description
0	SSE3	**Streaming SIMD Extensions 3 (SSE3)**. A value of 1 indicates the processor supports this technology.
1	PCLMULQDQ	A value of 1 indicates the processor supports PCLMULQDQ instruction.
2	DTES64	**64-bit DS Area**. A value of 1 indicates the processor supports DS area using 64-bit layout.
3	MONITOR	**MONITOR/MWAIT**. A value of 1 indicates the processor supports this feature.

Table 2-23. Feature Information Returned in the ECX Register (Continued)

Bit #	Mnemonic	Description
4	DS-CPL	**CPL Qualified Debug Store**. A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	Virtual Machine Extensions. A value of 1 indicates that the processor supports this technology.
6	SMX	Safer Mode Extensions. A value of 1 indicates that the processor supports this technology. See Chapter 6, "Safer Mode Extensions Reference".
7	EST	**Enhanced Intel SpeedStep® technology**. A value of 1 indicates that the processor supports this technology.
8	TM2	**Thermal Monitor 2**. A value of 1 indicates whether the processor supports this technology.
9	SSSE3	A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor.
10	CNXT-ID	**L1 Context ID**. A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
11	Reserved	Reserved
12	FMA	A value of 1 indicates the processor supports FMA extensions using YMM state.
13	CMPXCHG16B	**CMPXCHG16B Available**. A value of 1 indicates that the feature is available.
14	xTPR Update Control	**xTPR Update Control**. A value of 1 indicates that the processor supports changing IA32_MISC_ENABLES[bit 23].
15	PDCM	**Perfmon and Debug Capability**: A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	**Process-context identifiers.** A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4.2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.

Table 2-23. Feature Information Returned in the ECX Register (Continued)

Bit #	Mnemonic	Description
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	Reserved	Reserved
25	AES	A value of 1 indicates that the processor supports the AES instruction.
26	XSAVE	A value of 1 indicates that the processor supports the XFEATURE_ENABLED_MASK register and XSAVE/XRSTOR/XSETBV/XGETBV instructions to manage processor extended states.
27	OSXSAVE	A value of 1 indicates that the OS has enabled support for using XGETBV/XSETBV instructions to query processor extended states.
28	AVX	A value of 1 indicates that processor supports AVX instructions operating on 256-bit YMM state, and three-operand encoding of 256-bit and 128-bit SIMD instructions.
29	F16C	A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.
30	RDRND	A value of 1 indicates that processor supports RDRAND instruction.
31	Not Used	Always return 0

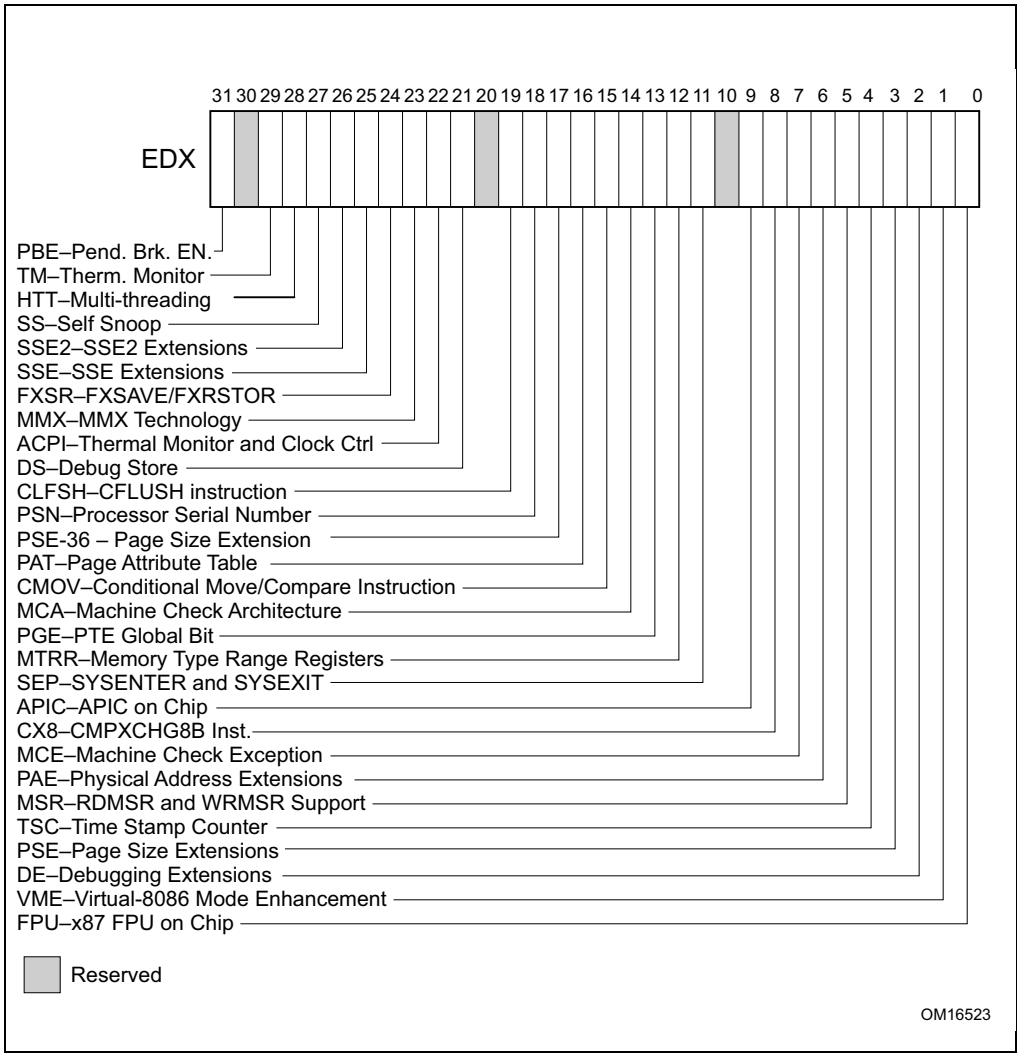


Figure 2-4. Feature Information Returned in the EDX Register

Table 2-24. More on Feature Information Returned in the EDX Register

Bit #	Mnemonic	Description
0	FPU	**floating-point Unit On-Chip.** The processor contains an x87 FPU.
1	VME	**Virtual 8086 Mode Enhancements.** Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
2	DE	**Debugging Extensions.** Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.
3	PSE	**Page Size Extension.** Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
4	TSC	**Time Stamp Counter.** The RDTSC instruction is supported, including CR4.TSD for controlling privilege.
5	MSR	**Model Specific Registers RDMSR and WRMSR Instructions.** The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.
6	PAE	**Physical Address Extension.** Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1. The actual number of address bits beyond 32 is not defined, and is implementation specific.
7	MCE	**Machine Check Exception**. Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.
8	CX8	**CMPXCHG8B Instruction.** The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
9	APIC	**APIC On-Chip.** The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).
10	Reserved	Reserved
11	SEP	**SYSENTER and SYSEXIT Instructions.** The SYSENTER and SYSEXIT and associated MSRs are supported.
12	MTRR	**Memory Type Range Registers.** MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.

Table 2-24. More on Feature Information Returned in the EDX Register(Continued)

Bit #	Mnemonic	Description
13	PGE	**PTE Global Bit.** The global bit in page directory entries (PDEs) and page table entries (PTEs) is supported, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.
14	MCA	**Machine Check Architecture.** The Machine Check Architecture, which provides a compatible mechanism for error reporting in P6 family, Pentium 4, Intel Xeon processors, and future processors, is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.
15	CMOV	**Conditional Move Instructions.** The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported
16	PAT	**Page Attribute Table.** Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory on a 4K granularity through a linear address.
17	PSE-36	**36-Bit Page Size Extension.** Extended 4-MByte pages that are capable of addressing physical memory beyond 4 GBytes are supported. This feature indicates that the upper four bits of the physical address of the 4-MByte page is encoded by bits 13-16 of the page directory entry.
18	PSN	**Processor Serial Number.** The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	**CLFLUSH Instruction.** CLFLUSH Instruction is supported.
20	Reserved	Reserved
21	DS	**Debug Store.** The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and precise event-based sampling (PEBS) facilities (see Chapter 16, "Debugging, Branch Profiles and Time-Stamp Counter," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).
22	ACPI	**Thermal Monitor and Software Controlled Clock Facilities.** The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	**Intel MMX Technology.** The processor supports the Intel MMX technology.
24	FXSR	**FXSAVE and FXRSTOR Instructions.** The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating-point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.

Table 2-24. More on Feature Information Returned in the EDX Register(Continued)

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

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

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

The encoding is as follows:

- The least-significant byte in register EAX (register AL) indicates the number of times the CPUID instruction must be executed with an input value of 2 to get a complete description of the processor's caches and TLBs. The first member of the family of Pentium 4 processors will return a 1.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. Table 2-25 shows the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache or TLB types. The descriptors may appear in any order.

Table 2-25. Encoding of Cache and TLB Descriptors

Descriptor Value	Cache or TLB Description
00H	Null descriptor
01H	Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries
02H	Instruction TLB: 4 MByte pages, 4-way set associative, 2 entries

Table 2-25. Encoding of Cache and TLB Descriptors (Continued)

Descriptor Value	Cache or TLB Description
03H	Data TLB: 4 KByte pages, 4-way set associative, 64 entries
04H	Data TLB: 4 MByte pages, 4-way set associative, 8 entries
05H	Data TLB1: 4 MByte pages, 4-way set associative, 32 entries
06H	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size
08H	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size
0AH	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size
0BH	Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries
0CH	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size
22H	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
25H	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
29H	3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
2CH	1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size
30H	1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size
40H	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size
42H	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size
43H	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size
44H	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size
45H	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size
46H	3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size
47H	3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size
49H	3rd-level cache: 4MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H); 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size
4AH	3rd-level cache: 6MByte, 12-way set associative, 64 byte line size
4BH	3rd-level cache: 8MByte, 16-way set associative, 64 byte line size
4CH	3rd-level cache: 12MByte, 12-way set associative, 64 byte line size
4DH	3rd-level cache: 16MByte, 16-way set associative, 64 byte line size

Table 2-25. Encoding of Cache and TLB Descriptors (Continued)

Descriptor Value	Cache or TLB Description
4EH	2nd-level cache: 6MByte, 24-way set associative, 64 byte line size
50H	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries
51H	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries
52H	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries
56H	Data TLB0: 4 MByte pages, 4-way set associative, 16 entries
57H	Data TLB0: 4 KByte pages, 4-way associative, 16 entries
5BH	Data TLB: 4 KByte and 4 MByte pages, 64 entries
5CH	Data TLB: 4 KByte and 4 MByte pages,128 entries
5DH	Data TLB: 4 KByte and 4 MByte pages,256 entries
60H	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size
66H	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size
67H	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size
68H	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size
70H	Trace cache: 12 K-μop, 8-way set associative
71H	Trace cache: 16 K-μop, 8-way set associative
72H	Trace cache: 32 K-μop, 8-way set associative
78H	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size
79H	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector
7DH	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size
7FH	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size
82H	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size
83H	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size
84H	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size
85H	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size
86H	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size
87H	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size
B0H	Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries

Table 2-25. Encoding of Cache and TLB Descriptors (Continued)

Descriptor Value	Cache or TLB Description
B1H	Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries
B3H	Data TLB: 4 KByte pages, 4-way set associative, 128 entries
B4H	Data TLB1: 4 KByte pages, 4-way associative, 256 entries
F0H	64-Byte prefetching
F1H	128-Byte prefetching

Example 2-1. Example of Cache and TLB Interpretation

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

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

Which means:

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

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

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

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 2-20.

The CPUID leaf 4 also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=4 and ECX=0 and use it as part of the topology enumeration algorithm described in Chapter 8, "Multiple-Processor Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

INPUT EAX = 5: Returns MONITOR and MWAIT Features

When CPUID executes with EAX set to 5, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 2-20.

INPUT EAX = 6: Returns Thermal and Power Management Features

When CPUID executes with EAX set to 6, the processor returns information about thermal and power management features. See Table 2-20.

INPUT EAX = 7: Returns Structured Extended Feature Enumeration Information

When CPUID executes with EAX set to 7 and ECX = 0, the processor returns information about the maximum number of sub-leaves that contain extended feature flags. See Table 2-20.

When CPUID executes with EAX set to 7 and ECX = n (n > 1and less than the number of non-zero bits in CPUID.(EAX=07H, ECX= 0H).EAX, the processor returns information about extended feature flags. See Table 2-20. In subleaf 0, only EAX has the number of subleafs. In subleaf 0, EBX, ECX & EDX all contain extended feature flags.

Table 2-26. Structured Extended Feature Leaf, Function 0, EBX Register

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

INPUT EAX = 9: Returns Direct Cache Access Information

When CPUID executes with EAX set to 9, the processor returns information about Direct Cache Access capabilities. See Table 2-20.

INPUT EAX = 10: Returns Architectural Performance Monitoring Features

When CPUID executes with EAX set to 10, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 2-20) is greater than Pn 0. See Table 2-20.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 16, "Debugging, Branch Profiles and Time-Stamp Counter," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

INPUT EAX = 11: Returns Extended Topology Information

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

INPUT EAX = 13: Returns Processor Extended States Enumeration Information

When CPUID executes with EAX set to 13 and ECX = 0, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 2-20.

When CPUID executes with EAX set to 13 and ECX = n (n > 1and less than the number of non-zero bits in CPUID.(EAX=0DH, ECX= 0H).EAX and CPUID.(EAX=0DH, ECX= 0H).EDX), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 2-20.

METHODS FOR RETURNING BRANDING INFORMATION

Use the following techniques to access branding information:

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

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: "Identification of Earlier IA-32 Processors" in Chapter 14 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The Processor Brand String Method

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

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

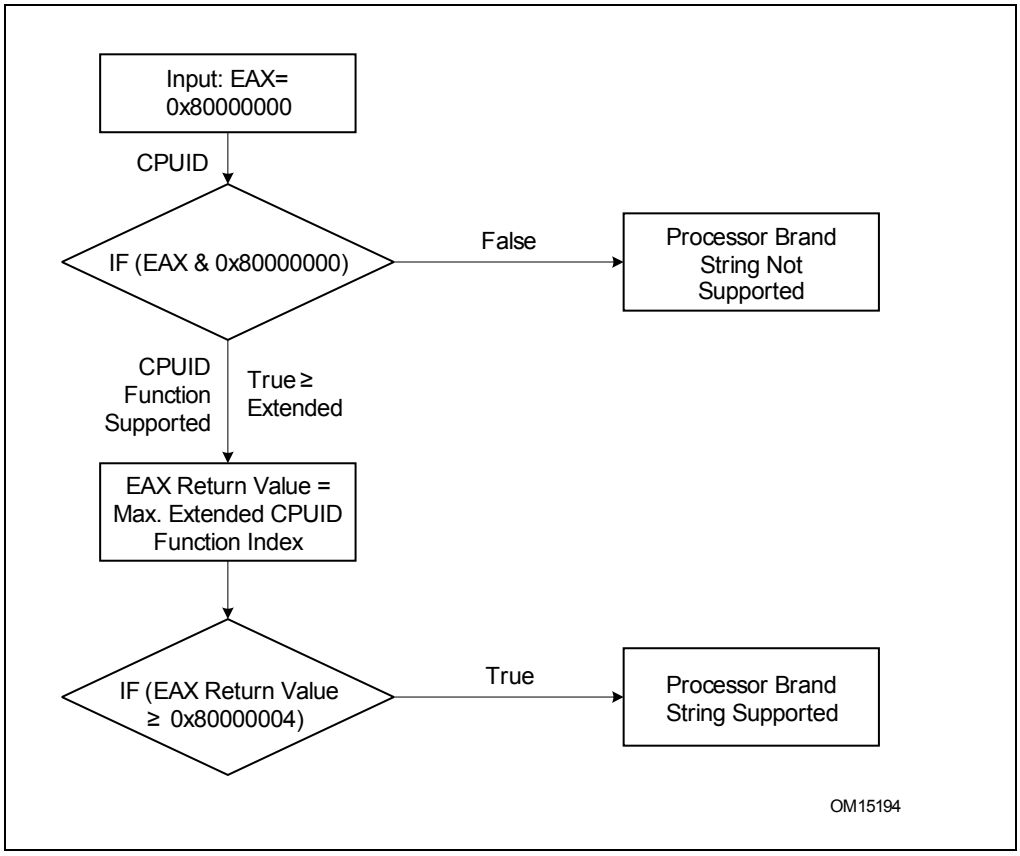


Figure 2-5. Determination of Support for the Processor Brand String

How Brand Strings Work

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

Table 2-27 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 2-27. Processor Brand String Returned with Pentium 4 Processor

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

Extracting the Maximum Processor Frequency from Brand Strings

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

NOTE

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

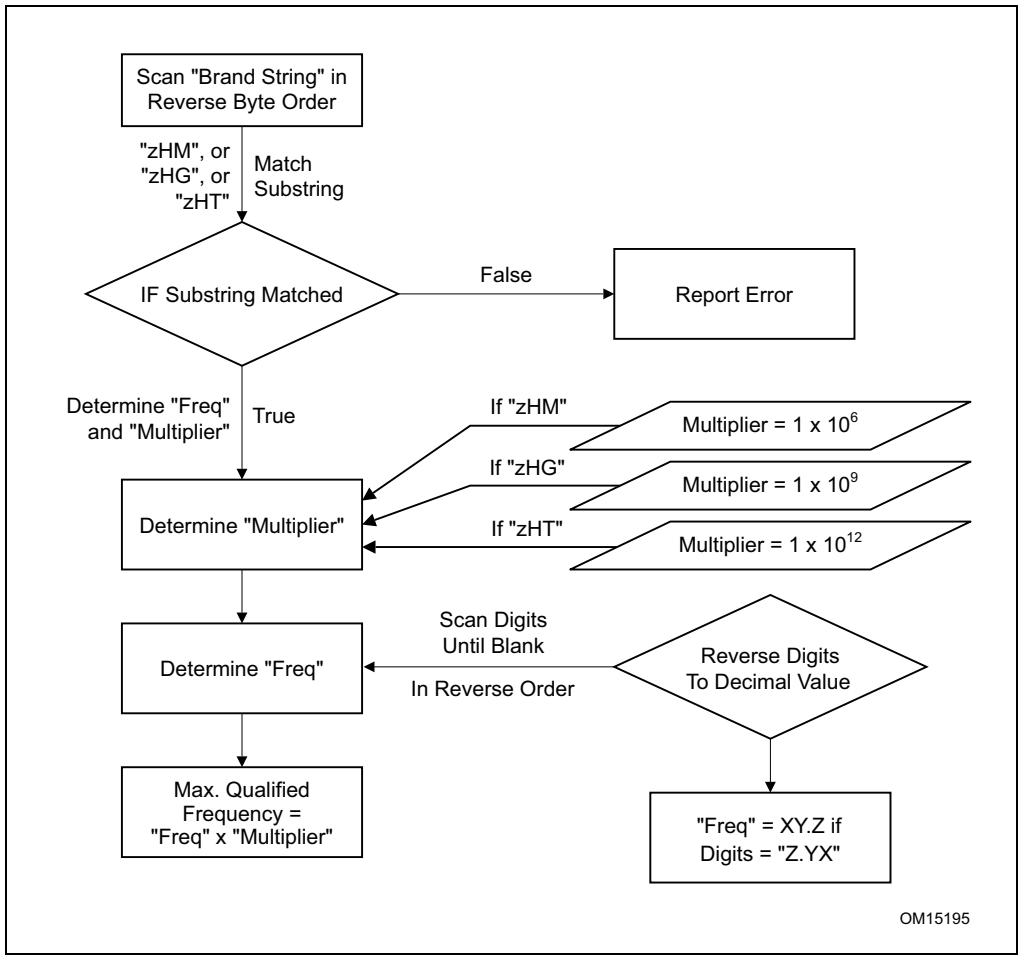


Figure 2-6. Algorithm for Extracting Maximum Processor Frequency

The Processor Brand Index Method

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

When CPUID executes with EAX set to 1, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that

do not support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 2-28 shows brand indices that have identification strings associated with them.

Table 2-28. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings

Brand Index	Brand String
00H	This processor does not support the brand identification feature
01H	Intel(R) Celeron(R) processor[1]
02H	Intel(R) Pentium(R) III processor[1]
03H	Intel(R) Pentium(R) III Xeon(R) processor; If processor signature = 000006B1h, then Intel(R) Celeron(R) processor
04H	Intel(R) Pentium(R) III processor
06H	Mobile Intel(R) Pentium(R) III processor-M
07H	Mobile Intel(R) Celeron(R) processor[1]
08H	Intel(R) Pentium(R) 4 processor
09H	Intel(R) Pentium(R) 4 processor
0AH	Intel(R) Celeron(R) processor[1]
0BH	Intel(R) Xeon(R) processor; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor MP
0CH	Intel(R) Xeon(R) processor MP
0EH	Mobile Intel(R) Pentium(R) 4 processor-M; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor
0FH	Mobile Intel(R) Celeron(R) processor[1]
11H	Mobile Genuine Intel(R) processor
12H	Intel(R) Celeron(R) M processor
13H	Mobile Intel(R) Celeron(R) processor[1]
14H	Intel(R) Celeron(R) processor
15H	Mobile Genuine Intel(R) processor
16H	Intel(R) Pentium(R) M processor
17H	Mobile Intel(R) Celeron(R) processor[1]
18H – 0FFH	RESERVED

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

IA-32 Architecture Compatibility

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

Operation

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

CASE (EAX) OF
    EAX = 0:
        EAX ← Highest basic function input value understood by CPUID;
        EBX ← Vendor identification string;
        EDX ← Vendor identification string;
        ECX ← Vendor identification string;
    BREAK;
    EAX = 1H:
        EAX[3:0] ← Stepping ID;
        EAX[7:4] ← Model;
        EAX[11:8] ← Family;
        EAX[13:12] ← Processor type;
        EAX[15:14] ← Reserved;
        EAX[19:16] ← Extended Model;
        EAX[27:20] ← Extended Family;
        EAX[31:28] ← Reserved;
        EBX[7:0] ← Brand Index; (* Reserved if the value is zero. *)
        EBX[15:8] ← CLFLUSH Line Size;
        EBX[16:23] ← Reserved; (* Number of threads enabled = 2 if MT enable fuse set. *)
        EBX[24:31] ← Initial APIC ID;
        ECX ← Feature flags; (* See Figure 2-3. *)
        EDX ← Feature flags; (* See Figure 2-4. *)
    BREAK;
    EAX = 2H:
        EAX ← Cache and TLB information;
        EBX ← Cache and TLB information;
        ECX ← Cache and TLB information;
        EDX ← Cache and TLB information;
    BREAK;
    EAX = 3H:
        EAX ← Reserved;
        EBX ← Reserved;
        ECX ← ProcessorSerialNumber[31:0];
        (* Pentium III processors only, otherwise reserved. *)
        EDX ← ProcessorSerialNumber[63:32];
        (* Pentium III processors only, otherwise reserved. *
```

```
BREAK
EAX = 4H:
    EAX ← Deterministic Cache Parameters Leaf; (* See Table 2-20. *)
    EBX ← Deterministic Cache Parameters Leaf;
    ECX ← Deterministic Cache Parameters Leaf;
    EDX ← Deterministic Cache Parameters Leaf;
BREAK;
EAX = 5H:
    EAX ← MONITOR/MWAIT Leaf; (* See Table 2-20. *)
    EBX ← MONITOR/MWAIT Leaf;
    ECX ← MONITOR/MWAIT Leaf;
    EDX ← MONITOR/MWAIT Leaf;
BREAK;
EAX = 6H:
    EAX ← Thermal and Power Management Leaf; (* See Table 2-20. *)
    EBX ← Thermal and Power Management Leaf;
    ECX ← Thermal and Power Management Leaf;
    EDX ← Thermal and Power Management Leaf;
BREAK;
EAX = 7H :
    EAX ← Structured Extended Feature Leaf; (* See Table 2-20. *);
    EBX ← Structured Extended Feature Leaf;
    ECX ← Structured Extended Feature Leaf;
    EDX ← Structured Extended Feature Leaf;
BREAK;
EAX = 8H:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = 9H:
    EAX ← Direct Cache Access Information Leaf; (* See Table 2-20. *)
    EBX ← Direct Cache Access Information Leaf;
    ECX ← Direct Cache Access Information Leaf;
    EDX ← Direct Cache Access Information Leaf;
BREAK;
EAX = AH:
    EAX ← Architectural Performance Monitoring Leaf; (* See Table 2-20. *)
    EBX ← Architectural Performance Monitoring Leaf;
    ECX ← Architectural Performance Monitoring Leaf;
    EDX ← Architectural Performance Monitoring Leaf;
    BREAK
```

```
    EAX = BH:
        EAX ← Extended Topology Enumeration Leaf; (* See Table 2-20. *)
        EBX ← Extended Topology Enumeration Leaf;
        ECX ← Extended Topology Enumeration Leaf;
        EDX ← Extended Topology Enumeration Leaf;
    BREAK;
    EAX = CH:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    EAX = DH:
        EAX ← Processor Extended State Enumeration Leaf; (* See Table 2-20. *)
        EBX ← Processor Extended State Enumeration Leaf;
        ECX ← Processor Extended State Enumeration Leaf;
        EDX ← Processor Extended State Enumeration Leaf;
    BREAK;
BREAK;
    EAX = 80000000H:
        EAX ← Highest extended function input value understood by CPUID;
        EBX ← Reserved;
        ECX ← Reserved;
        EDX ← Reserved;
    BREAK;
    EAX = 80000001H:
        EAX ← Reserved;
        EBX ← Reserved;
        ECX ← Extended Feature Bits (* See Table 2-20.*);
        EDX ← Extended Feature Bits (* See Table 2-20. *);
    BREAK;
    EAX = 80000002H:
        EAX ← Processor Brand String;
        EBX ← Processor Brand String, continued;
        ECX ← Processor Brand String, continued;
        EDX ← Processor Brand String, continued;
    BREAK;
    EAX = 80000003H:
        EAX ← Processor Brand String, continued;
        EBX ← Processor Brand String, continued;
        ECX ← Processor Brand String, continued;
        EDX ← Processor Brand String, continued;
    BREAK;
```

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

Flags Affected

None.

Exceptions (All Operating Modes)

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

§

This page was intentionally left blank.

CHAPTER 3
SYSTEM PROGRAMMING MODEL

This chapter describes the operating system programming considerations for AVX. The AES extension and PCLMULQDQ instruction follow the same system software requirements for XMM state support and SIMD floating-point exception support as SSE2, SSE3, SSSE3, SSE4 (see Chapter 12 of *IA-32 Intel Architecture Software Developer's Manual, Volumes 3A*).

The AVX and FMA extensions operate on 256-bit YMM registers, and require operating system to supports processor extended state management using XSAVE/XRSTOR instructions. VAESDEC/VAESDECLAST/VAESENC/VAESENCLAST/VAESIMC/VAESKEYGENASSIST/VPCLMULQDQ follow the same system programming requirements as AVX and FMA instructions operating on YMM states.

The basic requirements for an operating system using XSAVE/XRSTOR to manage processor extended states for current and future Intel Architecture processors can be found in Chapter 12 of *IA-32 Intel Architecture Software Developer's Manual, Volumes 3A*. This chapter covers additional requirements for OS to support YMM state.

3.1 YMM STATE, VEX PREFIX AND SUPPORTED OPERATING MODES

AVX and FMA instructions comprises of 256-bit and 128-bit instructions that operates on YMM states via VEX prefix encoding. SIMD instructions operating on XMM states (i.e. not accessing the upper 128 bits of YMM) generally do not use VEX prefix.

For processors that support YMM states, the YMM state exists in all operating modes. However, the available interfaces to access YMM states may vary in different modes. The processor's support for instruction extensions that employ VEX prefix encoding is independent of the processor's support for YMM state.

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

Note that bits 255:128 of YMM register state are maintained across transitions into and out of these modes. Because, XSAVE/XRSTOR instruction can operate in all operating modes, it is possible that the processor's YMM register state can be modified by software in any operating mode by executing XRSTOR. The YMM registers can be updated by XRSTOR using the state information stored in the XSAVE/XRSTOR area residing in memory.

3.2 YMM STATE MANAGEMENT

Operating systems must use the XSAVE/XRSTOR instructions for YMM state management. The XSAVE/XRSTOR instructions also provide flexible and efficient interface to manage XMM/MXCSR states and x87 FPU states in conjunction with new processor extended states.

An OS must enable its YMM state management to support AVX and FMA extensions. Otherwise, an attempt to execute an instruction in AVX or FMA extensions (including an enhanced 128-bit SIMD instructions using VEX encoding) will cause a #UD exception.

3.2.1 Detection of YMM State Support

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

3.2.2 Enabling of YMM State

An OS can enable YMM state support with the following steps:

- Verify the processor supports XSAVE/XRSTOR/XSETBV/XGETBV instructions and the XFEATURE_ENABLED_MASK register by checking CPUID.1.ECX.XSAVE[bit 26]=1.
- Verify the processor supports YMM state (i.e. bit 2 of XFEATURE_ENABLED_MASK is valid) by checking CPUID.(EAX=0DH, ECX=0):EAX.YMM[2]. The OS should also verify CPUID.(EAX=0DH, ECX=0):EAX.SSE[bit 1]=1, because the lower 128-bits of an YMM register are aliased to an XMM register.

  The OS must determine the buffer size requirement for the XSAVE area that will be used by XSAVE/XRSTOR (see CPUID instruction in Section 2.9).
- Set CR4.OSXSAVE[bit 18]=1 to enable the use of XSETBV/XGETBV instructions to write/read the XFEATURE_ENABLED_MASK register.
- Supply an appropriate mask via EDX:EAX to execute XSETBV to enable the processor state components that the OS wishes to manage using XSAVE/XRSTOR instruction. To enable x87 FPU, SSE and YMM state management using XSAVE/XRSTOR, the enable mask is EDX=0H, EAX=7H (The individual bits of XFEATURE_ENABLED_MASK is listed in Table 3-1).

To enable YMM state, the OS must use EDX:EAX[2:1] = 11B when executing XSETBV. An attempt to execute XSETBV with EDX:EAX[2:1] = 10B causes a #GP(0) exception.

Table 3-1. XFEATURE_ENABLED_MASK and Processor State Components

Bit	Meaning
0 - x87	If set, the processor supports x87 FPU state management via XSAVE/XRSTOR. This bit must be 1 if CPUID.01H:ECX.XSAVE[26] = 1.
1 - SSE	If set, the processor supports SSE state (XMM and MXCSR) management via XSAVE/XRSTOR. This bit must be set to '1' to enable AVX.
2 - YMM	If set, the processor supports YMM state (upper 128 bits of YMM registers) management via XSAVE. This bit must be set to '1' to enable AVX and FMA.

3.2.3 Enabling of SIMD Floating-Exception Support

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

The effect of CR4 setting that affects AVX and FMA enabling is listed in Table 3-2

Table 3-2. CR4 bits for AVX New Instructions technology support

Bit	Meaning
CR4.OSXSAVE[bit 18]	If set, the OS supports use of XSETBV/XGETBV instruction to access. the XFEATURE_ENABLED_MASK register, XSAVE/XRSTOR to manage processor extended state. Must be set to '1' to enable AVX and FMA.
CR4.OSXMMEXCPT[bit 10]	Must be set to 1 to enable SIMD floating-point exceptions. This applies to AVX, FMA operating on YMM states, and legacy 128-bit SIMD floating-point instructions operating on XMM states.
CR4.OSFXSR[bit 9]	Ignored by AVX and FMA instructions operating on YMM states. Must be set to 1 to enable SIMD instructions operating on XMM state.

3.2.4 The Layout of XSAVE Area

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

After the memory buff for XSAVE is allocated, the entire buffer must to cleared to zero prior to use by XSAVE.

For processors that support SSE and YMM states, the XSAVE area layout is listed in Table 3-3. The register fields of the first 512 byte of the XSAVE area are identical to those of the FXSAVE/FXRSTOR area.

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

Save Areas	Offset (Byte)	Size (Bytes)
FPU/SSE SaveArea	0	512
Header	512	64
Ext_Save_Area_2 (YMM)	CPUID.(EAX=0DH, ECX=2):EBX	CPUID.(EAX=0DH, ECX=2):EAX

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

Table 3-4. XSAVE Header Format

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

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

Note in general that the layout of the XSAVE/XRSTOR save area is fixed and may contain non-contiguous individual save area (Ext_Save_Area_X). The XSAVE/XRSTOR area is not compacted if some processor extended state features are not saved or are not supported by the processor and/or by system software.

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

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

3.2.5 XSAVE/XRSTOR Interaction with YMM State and MXCSR

The processor's action as a result of executing XRSTOR, on the MXCSR, XMM and YMM registers, are listed in Table 3-6 (Both bit 1 and bit 2 of the XFEATURE_ENABLED_MASK register are presumed to be 1). The XMM registers may be initialized by the processor (See XRSTOR operation in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*). When the MXCSR register is updated from memory, reserved bit checking is enforced. The saving/restoring of MXCSR is bound to both the SSE state and YMM state.

Table 3-6. XRSTOR Action on MXCSR, XMM Registers, YMM Registers

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

The processor supplied init values for each processor state component used by XRSTOR is listed in Table 3-7.

Table 3-7. Processor Supplied Init Values XRSTOR May Use

Processor State Component	Processor Supplied Register Values
x87 FPU State	FCW ← 037FH; FTW ← 0FFFFH; FSW ← 0H; FPU CS ← 0H; FPU DS ← 0H; FPU IP ← 0H; FPU DP ← 0; ST0-ST7 ← 0;
SSE State[1]	If 64-bit Mode: XMM0-XMM15 ← 0H; Else XMM0-XMM7 ← 0H
YMM State[1]	If 64-bit Mode: YMM0_H-YMM15_H ← 0H; Else YMM0_H-YMM7_H ← 0H

NOTES:

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

The action of XSAVE is listed in Table 3-8.

Table 3-8. XSAVE Action on MXCSR, XMM, YMM Register

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

3.2.6 Processor Extended State Save Optimization and XSAVEOPT

The XSAVEOPT instruction paired with XRSTOR is designed to provide a high performance method for system software to perform state save and restore.

A processor may indicate its support for the XSAVEOPT instruction if CPUID.(EAX=0DH, ECX=1):EAX.XSAVEOPT[Bit 0] = 1. The functionality of

XSAVEOPT is similar to XSAVE. Software can use XSAVEOPT/XRSTOR in a pair-wise manner similar to XSAVE/XRSTOR to save and restore processor extended states.

The syntax and operands for XSAVEOPT instructions are identical to XSAVE, i.e. the mask operand in EDX:EAX specifies the subset of enabled features to be saved.

Note that software using XSAVEOPT must observe the same restrictions as XSAVE while allocating a new save area. i.e., the header area must be initialized to zeroes. The first 64-bits in the save image header starting at offset 512 are referred to as XHEADER.BV. However, the instruction differs from XSAVE in several important aspects:

1. If a component state in the processor specified by the save mask corresponds to an INIT state, the instruction may clear the corresponding bit in XHEADER.BV, but may not write out the state (unlike the XSAVE instruction, which always writes out the state).
2. If the processor determines that the component state specified by the save mask hasn't been modified since the last XRSTOR, the instruction may not write out the state to the save area.
3. A implication of this optimization is that software which needs to examine the saved image must first check the XHEADER.BV to see if any bits are clear. If the header bit is clear, it means that the state is INIT and the saved memory image may not correspond to the actual processor state.
4. The performance of XSAVEOPT will always be better than or at least equal to that of XSAVE.

3.2.6.1 XSAVEOPT Usage Guidelines

When using the XSAVEOPT facility, software must be aware of the following guidelines:

1. The processor uses a tracking mechanism to determine which state components will be written to memory by the XSAVEOPT instruction. The mechanism includes three sub-conditions that are recorded internally each time XRSTOR is executed and evaluated on the invocation of the next XSAVEOPT. If a change is detected in any one of these sub-conditions, XSAVEOPT will behave exactly as XSAVE. The three sub-conditions are:
   - current CPL of the logical processor
   - indication whether or not the logical processor is in VMX non-root operation
   - linear address of the XSAVE/XRSTOR area
2. Upon allocation of a new XSAVE/XRSTOR area and before an XSAVE or XSAVEOPT instruction is used, the save area header (HEADER.XSTATE) must be initialized to zeroes for proper operation.
3. XSAVEOPT is designed primarily for use in context switch operations. The values stored by the XSAVEOPT instruction depend on the values previously stored in a given XSAVE area.

4. Manual modifications to the XSAVE area between an XRSTOR instruction and the matching XSAVEOPT may result in data corruption.
5. For optimization to be performed properly, the XRSTOR XSAVEOPT pair must use the same segment when referencing the XSAVE area and the base of that segment must be unchanged between the two operations.
6. Software should avoid executing XSAVEOPT into a buffer from which it hadn't previously executed a XRSTOR. For newly allocated buffers, software can execute XRSTOR with the linear address of the buffer and a restore mask of EDX:EAX = 0. Executing XRSTOR(0:0) doesn't restore any state, but ensures expected operation of the XSAVEOPT instruction.
7. The XSAVE area can be moved or even paged, but the contents at the linear address of the save area at an XSAVEOPT must be the same as that when the previous XRSTOR was performed.

A destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) will result in a general-protection (#GP) exception being generated. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

3.3 RESET BEHAVIOR

At processor reset

- YMM0-16 bits[255:0] are set to zero.
- XFEATURE_ENABLED_MASK[2:1] is set to zero, XFEATURE_ENABLED_MASK[0] is set to 1.
- CR4.OSXSAVE[bit 18] (and its mirror CPUID.1.ECX.OSXSAVE[bit 27]) is set to 0.

3.4 EMULATION

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

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

3.5 WRITING AVX FLOATING-POINT EXCEPTION HANDLERS

AVX and FMA floating-point exceptions are handled in an entirely analogous way to Legacy SSE floating-point exceptions. To handle unmasked SIMD floating-point exceptions, the operating system or executive must provide an exception handler.

The section titled "SSE and SSE2 SIMD Floating-Point Exceptions" in Chapter 11, "Programming with Streaming SIMD Extensions 2 (SSE2)," of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1,* describes the SIMD floating-point exception classes and gives suggestions for writing an exception handler to handle them.

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

This page was intentionally left blank.

§

CHAPTER 4
INSTRUCTION FORMAT

AVX and FMA instructions are encoded using a more efficient format than previous instruction extensions in the Intel 64 and IA-32 architecture. The improved encoding format make use a new prefix referred to as "VEX". The VEX prefix may be two or three bytes long, depending on the instruction semantics. Despite the length of the VEX prefix, the instruction encoding format using VEX addresses two important issues: (a) there exists inefficiency in instruction encoding due to SIMD prefixes and some fields of the REX prefix, (b) Both SIMD prefixes and REX prefix increase in instruction byte-length. This chapter describes the instruction encoding format using VEX.

4.1 INSTRUCTION FORMATS

Legacy instruction set extensions in IA-32 architecture employs one or more "single-purpose" byte as an "escape opcode", or required SIMD prefix (66H, F2H, F3H) to expand the processing capability of the instruction set. Intel 64 architecture uses the REX prefix to expand the encoding of register access in instruction operands. Both SIMD prefixes and REX prefix carry the side effect that they can cause the length of an instruction to increase significantly. Legacy Intel 64 and IA-32 instruction set are limited to supporting instruction syntax of only two operands that can be encoded to access registers (and only one can access a memory address).

Instruction encoding using VEX prefix provides several advantages:

- Instruction syntax support for three operands and up-to four operands when necessary. For example, the third source register used by VBLENDVPD is encoded using bits 7:4 of the immediate byte.
- Encoding support for vector length of 128 bits (using XMM registers) and 256 bits (using YMM registers)
- Encoding support for instruction syntax of non-destructive source operands.
- Elimination of escape opcode byte (0FH), SIMD prefix byte (66H, F2H, F3H) via a compact bit field representation within the VEX prefix.
- Elimination of the need to use REX prefix to encode the extended half of general-purpose register sets (R8-R15) for direct register access, memory addressing, or accessing XMM8-XMM15 (including YMM8-YMM15).
- Flexible and more compact bit fields are provided in the VEX prefix to retain the full functionality provided by REX prefix. REX.W, REX.X, REX.B functionalities are provided in the three-byte VEX prefix only because only a subset of SIMD instructions need them.
- Extensibility for future instruction extensions without significant instruction length increase.

Figure 4-1 shows the Intel 64 instruction encoding format with VEX prefix support. Legacy instruction without a VEX prefix is fully supported and unchanged. The use of VEX prefix in an Intel 64 instruction is optional, but a VEX prefix is required for Intel 64 instructions that operate on YMM registers or support three and four operand syntax. VEX prefix is not a constant-valued, "single-purpose" byte like 0FH, 66H, F2H, F3H in legacy SSE instructions. VEX prefix provides substantially richer capability than the REX prefix.

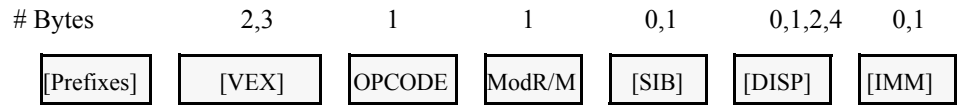


Figure 4-1. Instruction Encoding Format with VEX Prefix

4.1.1 VEX and the LOCK prefix

Any VEX-encoded instruction with a LOCK prefix preceding VEX will #UD.

4.1.2 VEX and the 66H, F2H, and F3H prefixes

Any VEX-encoded instruction with a 66H, F2H, or F3H prefix preceding VEX will #UD.

4.1.3 VEX and the REX prefix

Any VEX-encoded instruction with a REX prefix proceeding VEX will #UD.

4.1.4 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). The two-byte VEX is used mainly for 128-bit, scalar, and the most common 256-bit AVX instructions; while the three-byte VEX provides a compact replacement of REX and 3-byte opcode instructions (including AVX and FMA instructions). Beyond the first byte of the VEX prefix, it consists of a number of bit fields providing specific capability, they are shown in Figure 4-2.

The bit fields of the VEX prefix can be summarized by its functional purposes:

- Non-destructive source register encoding (applicable to three and four operand syntax): This is the first source operand in the instruction syntax. It is represented by the notation, VEX.vvvv. This field is encoded using 1's

complement form (inverted form), i.e. XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.

- Vector length encoding: This 1-bit field represented by the notation VEX.L. L= 0 means vector length is 128 bits wide, L=1 means 256 bit vector. The value of this field is written as VEX.128 or VEX.256 in this document to distinguish encoded values of other VEX bit fields.
- REX prefix functionality: Full REX prefix functionality is provided in the three-byte form of VEX prefix. However the VEX bit fields providing REX functionality are encoded using 1's complement form, i.e. XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.
  - Two-byte form of the VEX prefix only provides the equivalent functionality of REX.R, using 1's complement encoding. This is represented as VEX.R.
  - Three-byte form of the VEX prefix provides REX.R, REX.X, REX.B functionality using 1's complement encoding and three dedicated bit fields represented as VEX.R, VEX.X, VEX.B.
  - Three-byte form of the VEX prefix provides the functionality of REX.W only to specific instructions that need to override default 32-bit operand size for a general purpose register to 64-bit size in 64-bit mode. For those applicable instructions, VEX.W field provides the same functionality as REX.W. VEX.W field can provide completely different functionality for other instructions.

  Consequently, the use of REX prefix with VEX encoded instructions is not allowed. However, the intent of the REX prefix for expanding register set is reserved for future instruction set extensions using VEX prefix encoding format.
- Compaction of SIMD prefix: Legacy SSE instructions effectively use SIMD prefixes (66H, F2H, F3H) as an opcode extension field. VEX prefix encoding allows the functional capability of such legacy SSE instructions (operating on XMM registers, bits 255:128 of corresponding YMM unmodified) to be encoded using the VEX.pp field without the presence of any SIMD prefix. The VEX-encod 128-bit instruction will zero-out bits 255:128 of the destination register. VEX-encoded instruction may have 128 bit vector length or 256 bits length.
- Compaction of two-byte and three-byte opcode: More recently introduced legacy SSE instructions employ two and three-byte opcode. The one or two leading bytes are: 0FH, and 0FH 3AH/0FH 38H. The one-byte escape (0FH) and two-byte escape (0FH 3AH, 0FH 38H) can also be interpreted as an opcode extension field. The VEX.mmmmm field provides compaction to allow many legacy instruction to be encoded without the constant byte sequence, 0FH, 0FH 3AH, 0FH 38H. These VEX-encoded instruction may have 128 bit vector length or 256 bits length.

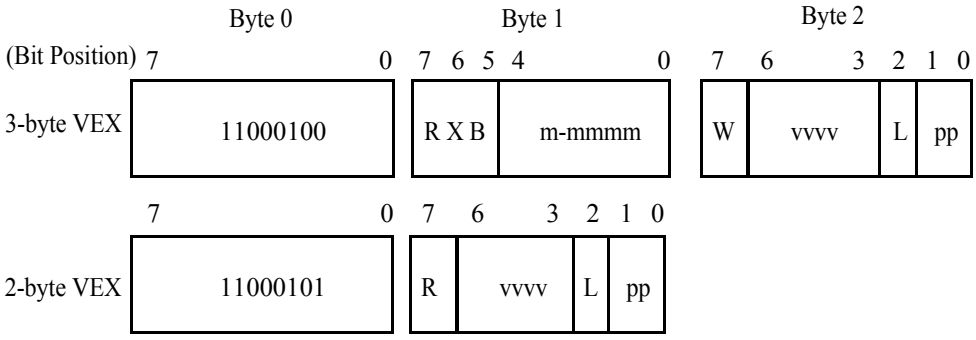
The VEX prefix is required to be the last prefix and immediately precedes the opcode bytes. It must follow any other prefixes. If VEX prefix is present a REX prefix is not supported.

The 3-byte VEX leaves room for future expansion with 3 reserved bits. REX and the 66h/F2h/F3h prefixes are reclaimed for future use.

VEX prefix has a two-byte form and a three byte form. If an instruction syntax can be encoded using the two-byte form, it can also be encoded using the three byte form of VEX. The latter increases the length of the instruction by one byte. This may be helpful in some situations for code alignment.

The VEX prefix supports 256-bit versions of floating-point SSE, SSE2, SSE3, and SSE4 instructions. Some additional support for 128-bit vector integer instructions is provided in Table A-1 of Appendix A. Note, certain new instruction functionality can only be encoded with the VEX prefix (See Appendix A, Table A-2).

The VEX prefix will #UD on any instruction containing MMX register sources or destinations.



R: REX.R in 1's complement (inverted) form
1: Same as REX.R=0 (must be 1 in 32-bit mode)
0: Same as REX.R=1 (64-bit mode only)

X: REX.X in 1's complement (inverted) form
1: Same as REX.X=0 (must be 1 in 32-bit mode)
0: Same as REX.X=1 (64-bit mode only)

B: REX.B in 1's complement (inverted) form
1: Same as REX.B=0 (Ignored in 32-bit mode).
0: Same as REX.B=1 (64-bit mode only)

W: opcode specific (use like REX.W, or used for opcode extension, or ignored, depending on the opcode byte)

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)

vvvv: a register specifier (in 1's complement form) or 1111 if unused.

L: Vector Length
0: scalar or 128-bit vector
1: 256-bit vector

pp: opcode extension providing equivalent functionality of a SIMD prefix
00: None
01: 66
10: F3
11: F2

Figure 4-2. VEX bitfields

The following subsections describe the various fields in two or three-byte VEX prefix:

4.1.4.1 VEX Byte 0, bits[7:0]

VEX Byte 0, bits [7:0] must contain the value 11000101b (C5h) or 11000100b (C4h). The 3-byte VEX uses the C4h first byte, while the 2-byte VEX uses the C5h first byte.

4.1.4.2 VEX Byte 1, bit [7] - 'R'

VEX Byte 1, bit [7] contains a bit analogous to a bit inverted REX.R. In protected and compatibility modes the bit must be set to '1' otherwise the instruction is LES or LDS.

This bit is present in both 2- and 3-byte VEX prefixes.

The usage of WRXB bits for legacy instructions is explained in detail section 2.2.1.2 of Intel 64 and IA-32 Architectures Software developer's manual, Volume 2A.

This bit is stored in bit inverted format.

4.1.4.3 3-byte VEX byte 1, bit[6] - 'X'

Bit[6] of the 3-byte VEX byte 1 encodes a bit analogous to a bit inverted REX.X. It is an extension of the SIB Index field in 64-bit modes. In 32-bit modes, this bit must be set to '1' otherwise the instruction is LES or LDS.

This bit is available only in the 3-byte VEX prefix.

This bit is stored in bit inverted format.

4.1.4.4 3-byte VEX byte 1, bit[5] - 'B'

Bit[5] of the 3-byte VEX byte 1 encodes a bit analogous to a bit inverted REX.B. In 64-bit modes, it is an extension of the ModR/M r/m field, or the SIB base field. In 32-bit modes, this bit is ignored.

This bit is available only in the 3-byte VEX prefix.

This bit is stored in bit inverted format.

4.1.4.5 3-byte VEX byte 2, bit[7] - 'W'

Bit[7] of the 3-byte VEX byte 2 is represented by the notation VEX.W. It can provide following functions, depending on the specific opcode.

- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have a general-purpose register operand with its operand size attribute promotable by REX.W), if REX.W promotes the operand size attribute of the general-purpose register operand in legacy SSE instruction, VEX.W has same meaning in the corresponding AVX equivalent form. In 32-bit

modes, VEX.W is silently ignored.
- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have operands with their operand size attribute fixed and not promotable by REX.W), if REX.W is don't care in legacy SSE instruction, VEX.W is ignored in the corresponding AVX equivalent form irrespective of mode.
- For new AVX instructions where VEX.W has no defined function (typically these meant the combination of the opcode byte and VEX.mmmmm did not have any equivalent SSE functions), VEX.W is reserved as zero and setting to other than zero will cause instruction to #UD.

4.1.4.6 2-byte VEX Byte 1, bits[6:3] and 3-byte VEX Byte 2, bits [6:3]- 'vvvv' the Source or dest Register Specifier

In 32-bit mode the VEX first byte C4 and C5 alias onto the LES and LDS instructions. To maintain compatibility with existing programs the VEX 2nd byte, bits [7:6] must be 11b. To achieve this, the VEX payload bits are selected to place only inverted, 64-bit valid fields (extended register selectors) in these upper bits.

The 2-byte VEX Byte 1, bits [6:3] and the 3-byte VEX, Byte 2, bits [6:3] encode a field (shorthand VEX.vvvv) that for instructions with 2 or more source registers and an XMM or YMM or memory destination encodes the first source register specifier stored in inverted (1's complement) form.

VEX.vvvv is not used by the instructions with one source (except certain shifts, see below) or on instructions with no XMM or YMM or memory destination. If an instruction does not use VEX.vvvv then it should be set to 1111b otherwise instruction will #UD.

In 64-bit mode all 4 bits may be used. See Table 4-1 for the encoding of the XMM or YMM registers. In 32-bit and 16-bit modes bit 6 must be 1 (if bit 6 is not 1, the 2-byte VEX version will generate LDS instruction and the 3-byte VEX version will ignore this bit).

Table 4-1. VEX.vvvv to register name mapping

VEX.vvvv	Dest Register	Valid in Legacy/Compatibility 32-bit modes?
1111B	XMM0/YMM0	Valid
1110B	XMM1/YMM1	Valid
1101B	XMM2/YMM2	Valid
1100B	XMM3/YMM3	Valid
1011B	XMM4/YMM4	Valid
1010B	XMM5/YMM5	Valid
1001B	XMM6/YMM6	Valid
1000B	XMM7/YMM7	Valid
0111B	XMM8/YMM8	Invalid
0110B	XMM9/YMM9	Invalid
0101B	XMM10/YMM10	Invalid
0100B	XMM11/YMM11	Invalid
0011B	XMM12/YMM12	Invalid
0010B	XMM13/YMM13	Invalid
0001B	XMM14/YMM14	Invalid
0000B	XMM15/YMM15	Invalid

The VEX.vvvv field is encoded in bit inverted format for accessing a register operand.

4.1.5 Instruction Operand Encoding and VEX.vvvv, ModR/M

VEX-encoded instructions support three-operand and four-operand instruction syntax. Some VEX-encoded instructions have syntax with less than three operands, e.g. VEX-encoded pack shift instructions support one source operand and one destination operand).

The roles of VEX.vvvv, reg field of ModR/M byte (ModR/M.reg), r/m field of ModR/M byte (ModR/M.r/m) with respect to encoding destination and source operands vary with different type of instruction syntax.

The role of VEX.vvvv can be summarized to three situations:

- VEX.vvvv encodes the first source register operand, specified in inverted (1's complement) form and is valid for instructions with 2 or more source operands.
- VEX.vvvv encodes the destination register operand, specified in 1's complement form for certain vector shifts. The instructions where VEX.vvvv is used as a destination are listed in Table 4-2. The notation in the "Opcode" column in Table 4-2 is described in detail in section 5.1.1

- VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b.

Table 4-2. Instructions with a VEX.vvvv destination

Opcode	Instruction mnemonic
VEX.NDD.128.66.0F 73 /7 ib	VPSLLDQ xmm1, xmm2, imm8
VEX.NDD.128.66.0F 73 /3 ib	VPSRLDQ xmm1, xmm2, imm8
VEX.NDD.128.66.0F 71 /2 ib	VPSRLW xmm1, xmm2, imm8
VEX.NDD.128.66.0F 72 /2 ib	VPSRLD xmm1, xmm2, imm8
VEX.NDD.128.66.0F 73 /2 ib	VPSRLQ xmm1, xmm2, imm8
VEX.NDD.128.66.0F 71 /4 ib	VPSRAW xmm1, xmm2, imm8
VEX.NDD.128.66.0F 72 /4 ib	VPSRAD xmm1, xmm2, imm8
VEX.NDD.128.66.0F 71 /6 ib	VPSLLW xmm1, xmm2, imm8
VEX.NDD.128.66.0F 72 /6 ib	VPSLLD xmm1, xmm2, imm8
VEX.NDD.128.66.0F 73 /6 ib	VPSLLQ xmm1, xmm2, imm8

The role of ModR/M.r/m field can be summarized to two situations:

- ModR/M.r/m encodes the instruction operand that references a memory address.
- For some instructions that do not support memory addressing semantics, ModR/M.r/m encodes either the destination register operand or a source register operand.

The role of ModR/M.reg field can be summarized to two situations:

- ModR/M.reg encodes either the destination register operand or a source register operand.
- For some instructions, ModR/M.reg is treated as an opcode extension and not used to encode any instruction operand.

For instruction syntax that support four operands, VEX.vvvv, ModR/M.r/m, ModR/M.reg encodes three of the four operands. The role of bits 7:4 of the immediate byte serves the following situation:

- Imm8[7:4] encodes the third source register operand.

4.1.5.1 3-byte VEX byte 1, bits[4:0] - "m-mmmm"

Bits[4:0] of the 3-byte VEX byte 1 encode an implied leading opcode byte (0F, 0F 38, or 0F 3A). Several bits are reserved for future use and will #UD unless 0.

Table 4-3. VEX.m-mmmm interpretation

VEX.m-mmmm	Implied Leading Opcode Bytes
00000B	Reserved
00001B	0F
00010B	0F 38
00011B	0F 3A
00100-11111B	Reserved
(2-byte VEX)	0F

VEX.m-mmmm is only available on the 3-byte VEX. The 2-byte VEX implies a leading 0Fh opcode byte.

4.1.5.2 2-byte VEX byte 1, bit[2], and 3-byte VEX byte 2, bit [2]- "L"

The vector length field, VEX.L, is encoded in bit[2] of either the second byte of 2-byte VEX, or the third byte of 3-byte VEX. If "VEX.L = 1", it indicates 256-bit vector operation. "VEX.L = 0" indicates scalar and 128-bit vector operations.

The instruction VZEROUPPER is a special case that is encoded with VEX.L = 0, although its operation zero's bits 255:128 of all YMM registers accessible in the current operating mode.

See the following table.

Table 4-4. VEX.L interpretation

VEX.L	Vector Length
0	128-bit (or 32/64-bit scalar)
1	256-bit

4.1.5.3 2-byte VEX byte 1, bits[1:0], and 3-byte VEX byte 2, bits [1:0]- "pp"

Up to one implied prefix is encoded by bits[1:0] of either the 2-byte VEX byte 1 or the 3-byte VEX byte 2. The prefix behaves as if it was encoded prior to VEX, but after all other encoded prefixes.

See the following table.

Table 4-5. VEX.pp interpretation

pp	Implies this prefix after other prefixes but before VEX
00B	None
01B	66
10B	F3
11B	F2

4.1.6 The Opcode Byte

One (and only one) opcode byte follows the 2 or 3 byte VEX. Legal opcodes are specified in Appendix B, in color. Any instruction that uses illegal opcode will #UD.

4.1.7 The MODRM, SIB, and Displacement Bytes

The encodings are unchanged but the interpretation of reg_field or rm_field differs (see above).

4.1.8 The Third Source Operand (Immediate Byte)

VEX-encoded instructions can support instruction with a four operand syntax. VBLENDVPD, VBLENDVPS, and PBLENDVB use imm8[7:4] to encode one of the source registers.

4.1.9 AVX Instructions and the Upper 128-bits of YMM registers

If an instruction with a destination XMM register is encoded with a VEX prefix, the processor zeroes the upper 128 bits of the equivalent YMM register. Legacy SSE instructions without VEX preserve the upper 128-bits.

4.1.10 AVX Instruction Length

The AVX and FMA instructions described in this document (including VEX and ignoring other prefixes) do not exceed 11 bytes in length, but may increase in the future. The maximum length of an Intel 64 and IA-32 instruction remains 15 bytes.

This page was intentionally left blank.

§

CHAPTER 5
INSTRUCTION SET REFERENCE

Instructions that are described in this document follow the general documentation convention established in *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A* and *2B*. Additional notations and conventions adopted in this document are listed in *Section 5.1. Section 5.2* covers supplemental information that applies to a specific subset of instructions.

5.1 INTERPRETING INSTRUCTION REFERENCE PAGES

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

5.1.1 Instruction Format

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

ADDSD- ADD Scalar Double-Precision Floating-Point Values (THIS IS AN EXAMPLE)

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 58 /r ADDSD xmm1, xmm2/m64	A	V/V	SSE2	Add the low double-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1
VEX.NDS.128.F2.0F.WIG 58 /r VADDSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Add the low double-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

5.1.2 Opcode Column in the Instruction Summary Table

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

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

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

- **VEX:** indicates the presence of the VEX prefix is required. The VEX prefix can be encoded using the three-byte form (the first byte is C4H), or using the two-byte form (the first byte is C5H). The two-byte form of VEX only applies to those instructions that do not require the following fields to be encoded:

VEX.mmmmm, VEX.W, VEX.X, VEX.B. Refer to Section 4.1.4 for more detail on the VEX prefix

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

— **NDS, NDD, DDS:** 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.

  • VEX.DDS: VEX.vvvv encodes the second source register in a three-operand instruction syntax where the content of first source register will be overwritten by the result.

  • If none of NDS, NDD, and DDS 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.

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

  • If VEX.256 is present in the opcode column: The semantics of the instruction must be encoded with VEX.L = 1. An attempt to encode this instruction with VEX.L= 0 can result in one of two situations: (a) if VEX.128 version is defined, the processor will behave according to the defined VEX.128 behavior; (b) an #UD occurs if there is no VEX.128 version defined.

  • If VEX.128 is present in the opcode column but there is no VEX.256 version defined for the same opcode byte: Two situations apply: (a) For VEX-encoded, 128-bit SIMD integer instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception; (b) For VEX-encoded, 128-bit packed floating-point instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception (e.g. VMOVLPS).

  • If VEX.LIG is present in the opcode column: The VEX.L value is ignored. This generally applies to VEX-encoded scalar SIMD floating-point instructions. Scalar SIMD floating-point instruction can be distinguished from the mnemonic of the instruction. Generally, the last two letters of the instruction mnemonic would be either "SS", "SD", or "SI" for SIMD floating-point conversion instructions.

  - If VEX.LZ is present in the opcode column: The VEX.L must be encoded to be 0B, an #UD occurs if VEX.L is not zero.

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

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

— **0F,0F3A,0F38 and 2-byte/3-byte VEX.** The presence of 0F3A and 0F38 in the opcode column implies that opcode can only be encoded by the three-byte form of VEX. The presence of 0F in the opcode column does not preclude the opcode to be encoded by the two-byte of VEX if the semantics of the opcode does not require any subfield of VEX not present in the two-byte form of the VEX prefix.

— **W0:** VEX.W=0.

— **W1:** VEX.W=1.

— 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. Please see Section 4.1.4 on the subfield definitions within VEX.

— **WIG:** can use C5H form or VEX.W value is ignored in the C4H form of VEX prefix.

— If WIG is present, the instruction may be encoded using either the two-byte form or the three-byte form of VEX. When encoding the instruction using the three-byte form of VEX, the value of VEX.W is ignored.

- **opcode:** Instruction opcode.
- **/is4:** An 8-bit immediate byte is present containing a source register specifier in imm[7:4] and instruction-specific payload in imm[3:0].

- **imz2**: Part of the is4 immediate byte providing control functions that apply to two-source permute instructions
- In general, the encoding o f VEX.R, VEX.X, VEX.B field are not shown explicitly in the opcode column. The encoding scheme of VEX.R, VEX.X, VEX.B fields must follow the rules defined in Section 4.1.4.

5.1.3 Instruction Column in the Instruction Summary Table

<additions to the eponymous PRM section>

- **ymm** — a YMM register. The 256-bit YMM registers are: YMM0 through YMM7**;** YMM8 through YMM15 are available in 64-bit mode.
- **m256** — A 32-byte operand in memory. This nomenclature is used only with AVX and FMA instructions.
- **ymm/m256** - a YMM register or 256-bit memory operand.
- **<YMM0>:** indicates use of the YMM0 register as an implicit argument.
- **SRC1 -** Denotes the first source operand in the instruction syntax of an instruction encoded with the VEX prefix and having two or more source operands.
- **SRC2 -** Denotes the second source operand in the instruction syntax of an instruction encoded with the VEX prefix and having two or more source operands.
- **SRC3 -** Denotes the third source operand in the instruction syntax of an instruction encoded with the VEX prefix and having three source operands.
- **SRC -** The source in a AVX single-source instruction or the source in a Legacy SSE instruction.
- **DST -** the destination in a AVX instruction. In Legacy SSE instructions can be either the destination, first source, or both. This field is encoded by reg_field.

5.1.4 Operand Encoding column in the Instruction Summary Table

The “operand encoding” column is abbreviated as Op/En in the Instruction Summary table heading. Each entry corresponds to a specific instruction syntax in the immediate column to its left and points to a corresponding row in a separate instruction operand encoding table immediately following the instruction summary table. The operand encoding table in each instruction reference page lists each instruction operand (according to each instruction syntax and operand ordering shown in the instruction column) relative to the ModRM byte, VEX.vvvv field or additional operand encoding placement.

5.1.5 64/32 bit Mode Support column in the Instruction Summary Table

The "64/32 bit Mode Support" column in the Instruction Summary table indicates whether an opcode sequence is supported in (a) 64-bit mode or (b) the Compatibility mode and other IA-32 modes that apply in conjunction with the CPUID feature flag associated specific instruction extensions.

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

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

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

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

5.1.6 CPUID Support column in the Instruction Summary Table

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

5.2 AES TRANSFORMATIONS AND DATA STRUCTURE

5.2.1 Little-Endian Architecture and Big-Endian Specification (FIPS 197)

FIPS 197 document defines the Advanced Encryption Standard (AES) and includes a set of test vectors for testing all of the steps in the algorithm, and can be used for testing and debugging.

The following observation is important for using the AES instructions offered in Intel 64 Architecture: FIPS 197 text convention is to write hex strings with the low-memory byte on the left and the high-memory byte on the right. Intel's convention is the reverse. It is similar to the difference between Big Endian and Little Endian notations.

In other words, a 128 bits vector in the FIPS document, when read from left to right, is encoded as [7:0, 15:8, 23:16, 31:24, …127:120]. Note that inside the byte, the encoding is [7:0], so the first bit from the left is the most significant bit. In practice, the test vectors are written in hexadecimal notation, where pairs of hexadecimal digits define the different bytes. To translate the FIPS 197 notation to an Intel 64 architecture compatible ("Little Endian") format, each test vector needs to be byte-reflected to [127:120,… 31:24, 23:16, 15:8, 7:0].

Example A:

FIPS Test vector: 0x000102030405060708090a0b0c0d0e0f

Intel AES Hardware: 0x0f0e0d0c0b0a09080706050403020100

It should be pointed out that the only thing at issue is a textual convention, and programmers do not need to perform byte-reversal in their code, when using the AES instructions.

5.2.1.1 AES Data Structure in Intel 64 Architecture

The AES instructions that are defined in this document operate on one or on two 128 bits source operands: State and Round Key. From the architectural point of view, the state is input in an xmm register and the Round key is input either in an xmm register or a 128-bit memory location.

In AES algorithm, the state (128 bits) can be viewed as 4 32-bit doublewords ("Word"s in AES terminology): X3, X2, X1, X0.

The state may also be viewed as a set of 16 bytes. The 16 bytes can also be viewed as a 4x4 matrix of bytes where S(i, j) with i, j = 0, 1, 2, 3 compose the 32-bit "word"s as follows:

X0 = S (3, 0) S (2, 0) S (1, 0) S (0, 0)

X1 = S (3, 1) S (2, 1) S (1, 1) S (0, 1)

X2 = S (3, 2) S (2, 2) S (1, 2) S (0, 2)

X3 = S (3, 3) S (2, 3) S (1, 3) S (0, 3)

The following tables, Table 5-1 through Table 5-4, illustrate various representations of a 128-bit state.

Table 5-1. Byte and 32-bit Word Representation of a 128-bit State

Byte #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit Position	**127 - 120**	**119 - 112**	**111 - 103**	**103 - 96**	**95 -88**	**87 -80**	**79 -72**	**71 -64**	**63 -56**	**55 -48**	**47 -40**	**39 -32**	**31 -24**	**23 -16**	**15 -8**	**7 - 0**
	127 - 96				95 - 64				64 - 32				31 - 0			
State Word	X3				X2				X1				X0			
State Byte	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A

Table 5-2. Matrix Representation of a 128-bit State

A	E	I	M	S(0, 0)	S(0, 1)	S(0, 2)	S(0, 3)
B	F	J	N	S(1, 0)	S(1, 1)	S(1, 2)	S(1, 3)
C	G	K	O	S(2, 0)	S(2, 1)	S(2, 2)	S(2, 3)
D	H	L	P	S(3, 0)	S(3, 1)	S(3, 2)	S(3, 3)

Example:

FIPS vector: d4 bf 5d 30 e0 b4 52 ae b8 41 11 f1 1e 27 98 e5

This vector has the “least significant” byte d4 and the significant byte e5 (written in Big Endian format in the FIPS document). When it is translated to IA notations, the encoding is:

Table 5-3. Little Endian Representation of a 128-bit State

Byte #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
State Byte	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A
State Value	e5	98	27	1e	f1	11	41	b8	ae	52	b4	e0	30	5d	bf	d4

Table 5-4. Little Endian Representation of a 4x4 Byte Matrix

A	E	I	M		d4	e0	b8	1e
B	F	J	N		bf	b4	41	27
C	G	K	O		5d	52	11	98

Table 5-4. Little Endian Representation of a 4x4 Byte Matrix

D	H	L	P		30	ae	f1	e5

5.2.2 AES Transformations and Functions

The following functions and transformations are used in the algorithmic descriptions of AES instruction extensions AESDEC, AESDECLAST, AESENC, AESENCLAST, AESIMC, AESKEYGENASSIST.

Note that these transformations are expressed here in a Little Endian format (and not as in the FIPS 197 document).

- MixColumns(): A byte-oriented 4x4 matrix transformation on the matrix representation of a 128-bit AES state. A FIPS-197 defined 4x4 matrix is multiplied to each 4x1 column vector of the AES state. The columns are considered polynomials with coefficients in the Finite Field that is used in the definition of FIPS 197, the operations ("multiplication" and "addition") are in that Finite Field, and the polynomials are reduced modulo x^4+1.

  The MixColumns() transformation defines the relationship between each byte of the result state, represented as S'(i, j) of a 4x4 matrix (see Section 5.2.1), as a function of input state bytes, S(i, j), as follows

  S'(0, j) ← FF_MUL(02H, S(0, j)) XOR FF_MUL(03H, S(1, j)) XOR S(2, j) XOR S(3, j)

  S'(1, j) ← S(0, j) XOR FF_MUL(02H, S(1, j)) XOR FF_MUL(03H, S(2, j)) XOR S(3, j)

  S'(2, j) ← S(0, j) XOR S(1, j) XOR FF_MUL(02H, S(2, j)) XOR FF_MUL(03H, S(3, j))

  S'(3, j) ← FF_MUL(03H, S(0, j)) XOR S(1, j) XOR S(2, j) XOR FF_MUL(02H, S(3, j))

  where j = 0, 1, 2, 3. FF_MUL(Byte1, Byte2) denotes the result of multiplying two elements (represented by Byte1 and byte2) in the Finite Field representation that defines AES. The result of produced bye FF_MUL(Byte1, Byte2) is an element in the Finite Field (represented as a byte). A Finite Field is a field with a finite number of elements, and when this number can be represented as a power of 2 (2n), its elements can be represented as the set of 2n binary strings of length n. AES uses a finite field with n=8 (having 256 elements). With this representation, "addition" of two elements in that field is a bit-wise XOR of their binary-string representation, producing another element in the field. Multiplication of two elements in that field is defined using an irreducible polynomial (for AES, this polynomial is $m(x) = x^8 + x^4 + x^3 + x + 1$). In this Finite Field representation, the bit value of bit position k of a byte represents the coefficient of a polynomial of order k, e.g., 1010_1101B (ADH) is represented by the polynomial $(x^7 + x^5 + x^3 + x^2 + 1)$. The byte value result of multiplication of two elements is obtained by a carry-less multiplication of the two corresponding polynomials, followed by reduction modulo the polynomial, where the remainder

is calculated using operations defined in the field. For example, FF_MUL(57H, 83H) = C1H, because the carry-less polynomial multiplication of the polynomials represented by 57H and 83H produces ($x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$), and the remainder modulo m(x) is ($x^7 + x^6 + 1$).

- RotWord(): performs a byte-wise cyclic permutation (rotate right in little-endian byte order) on a 32-bit AES word.

  The output word X'[j] of RotWord(X[j]) where X[j] represent the four bytes of column j, S(i, j), in descending order X[j] = (S(3, j), S(2, j), S(1, j), S(0, j)); X'[j] = (S'(3, j), S'(2, j), S'(1, j), S'(0, j)) ← (S(0, j), S(3, j), S(2, j), S(1, j))

- ShiftRows(): A byte-oriented matrix transformation that processes the matrix representation of a 16-byte AES state by cyclically shifting the last three rows of the state by different offset to the left, see Figure 5-5.

Table 5-5. The ShiftRows Transformation

Matrix Representation of Input State				Output of ShiftRows			
A	E	I	M	A	E	I	M
B	F	J	N	F	J	N	B
C	G	K	O	K	O	C	G
D	H	L	P	P	D	H	L

- SubBytes(): A byte-oriented transformation that processes the 128-bit AES state by applying a non-linear substitution table (S-BOX) on each byte of the state.

  The SubBytes() function defines the relationship between each byte of the result state S'(i, j) as a function of input state byte S(i, j), by

  S'(i, j) ← S-Box (S(i, j)[7:4], S(i, j)[3:0])

  where S-BOX(S[7:4], S[3:0]) represents a look-up operation on a 16x16 table to return a byte value, see Table 5-6.

Table 5-6. Look-up Table Associated with S-Box Transformation

		S[3:0]															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
S[7:4]	0	**63**	**7c**	**77**	**7b**	**f2**	**6b**	**6f**	**c5**	**30**	**01**	**67**	**2b**	**fe**	**d7**	**ab**	**76**
	1	**ca**	**82**	**c9**	**7d**	**fa**	**59**	**47**	**f0**	**ad**	**d4**	**a2**	**af**	**9c**	**a4**	**72**	**c0**
	2	**b7**	**fd**	**93**	**26**	**36**	**3f**	**f7**	**cc**	**34**	**a5**	**e5**	**f1**	**71**	**d8**	**31**	**15**
	3	**04**	**c7**	**23**	**c3**	**18**	**96**	**05**	**9a**	**07**	**12**	**80**	**e2**	**eb**	**27**	**b2**	**75**
	4	**09**	**83**	**2c**	**1a**	**1b**	**6e**	**5a**	**a0**	**52**	**3b**	**d6**	**b3**	**29**	**e3**	**2f**	**84**
	5	**53**	**d1**	**00**	**ed**	**20**	**fc**	**b1**	**5b**	**6a**	**cb**	**be**	**39**	**4a**	**4c**	**58**	**cf**
	6	**d0**	**ef**	**aa**	**fb**	**43**	**4d**	**33**	**85**	**45**	**f9**	**02**	**7f**	**50**	**3c**	**9f**	**a8**
	7	**51**	**a3**	**40**	**8f**	**92**	**9d**	**38**	**f5**	**bc**	**b6**	**da**	**21**	**10**	**ff**	**f3**	**d2**
	8	**cd**	**0c**	**13**	**ec**	**5f**	**97**	**44**	**17**	**c4**	**a7**	**7e**	**3d**	**64**	**5d**	**19**	**73**
	9	**60**	**81**	**4f**	**dc**	**22**	**2a**	**90**	**88**	**46**	**ee**	**b8**	**14**	**de**	**5e**	**0b**	**db**
	a	**e0**	**32**	**3a**	**0a**	**49**	**06**	**24**	**5c**	**c2**	**d3**	**ac**	**62**	**91**	**95**	**e4**	**79**
	b	**e7**	**c8**	**37**	**6d**	**8d**	**d5**	**4e**	**a9**	**6c**	**56**	**f4**	**ea**	**65**	**7a**	**ae**	**08**
	c	**ba**	**78**	**25**	**2e**	**1c**	**a6**	**b4**	**c6**	**e8**	**dd**	**74**	**1f**	**4b**	**bd**	**8b**	**8a**
	d	**70**	**3e**	**b5**	**66**	**48**	**03**	**f6**	**0e**	**61**	**35**	**57**	**b9**	**86**	**c1**	**1d**	**9e**
	e	**e1**	**f8**	**98**	**11**	**69**	**d9**	**8e**	**94**	**9b**	**1e**	**87**	**e9**	**ce**	**55**	**28**	**df**
	f	**8c**	**a1**	**89**	**0d**	**bf**	**e6**	**42**	**68**	**41**	**99**	**2d**	**0f**	**b0**	**54**	**bb**	**16**

- SubWord(): produces an output AES word (four bytes) from the four bytes of an input word using a non-linear substitution table (S-BOX).

  X'[j] = (S'(3, j), S'(2, j), S'(1, j), S'(0, j)) ← (S-Box (S(3, j)), S-Box(S(2, j)), S-Box(S(1, j)), S-Box(S(0, j)))

- InvMixColumns(): The inverse transformation of MixColumns().

  The InvMixColumns() transformation defines the relationship between each byte of the result state S'(i, j) as a function of input state bytes, S(i, j), by

  S'(0, j) ← FF_MUL(0eH, S(0, j)) XOR FF_MUL(0bH, S(1, j)) XOR FF_MUL(0dH, S(2, j)) XOR FF_MUL(09H, S(3, j))

  S'(1, j) ← FF_MUL(09H, S(0, j)) XOR FF_MUL(0eH, S(1, j)) XOR FF_MUL(0bH, S(2, j)) XOR FF_MUL(0dH, S(3, j))

  S'(2, j) ← FF_MUL(0dH, S(0, j)) XOR FF_MUL(09H, S(1, j)) XOR FF_MUL(0eH, S(2, j)) XOR FF_MUL(0bH, S(3, j))

S'(3, j) ← FF_MUL(0bH, S(0, j)) XOR FF_MUL(0dH, S(1, j)) XOR FF_MUL(09H, S(2, j)) XOR FF_MUL(0eH, S(3, j)), where j = 0, 1, 2, 3.

- InvShiftRows(): The inverse transformation of InvShiftRows(). The InvShiftRows() transforms the matrix representation of a 16-byte AES state by cyclically shifting the last three rows of the state by different offset to the right, see Table 5-7.

Table 5-7. The InvShiftRows Transformation

Matrix Representation of Input State				Output of ShiftRows			
A	E	I	M	A	E	I	M
B	F	J	N	N	B	F	J
C	G	K	O	K	O	C	G
D	H	L	P	H	L	P	D

- InvSubBytes(): The inverse transformation of SubBytes().

  The InvSubBytes() transformation defines the relationship between each byte of the result state S'(i, j) as a function of input state byte S(i, j), by

  S'(i, j) ← InvS-Box (S(i, j)[7:4], S(i, j)[3:0])

  where InvS-BOX(S[7:4], S[3:0]) represents a look-up operation on a 16x16 table to return a byte value, see Table 5-8.

Table 5-8. Look-up Table Associated with InvS-Box Transformation

		S[3:0]															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
S[7:4]	0	**52**	**09**	**6a**	**d5**	**30**	**36**	**a5**	**38**	**bf**	**40**	**a3**	**9e**	**81**	**f3**	**d7**	**fb**
	1	**7c**	**e3**	**39**	**82**	**9b**	**2f**	**ff**	**87**	**34**	**8e**	**43**	**44**	**c4**	**de**	**e9**	**cb**
	2	**54**	**7b**	**94**	**32**	**a6**	**c2**	**23**	**3d**	**ee**	**4c**	**95**	**0b**	**42**	**fa**	**c3**	**4e**
	3	**08**	**2e**	**a1**	**66**	**28**	**d9**	**24**	**b2**	**76**	**5b**	**a2**	**49**	**6d**	**8b**	**d1**	**25**
	4	**72**	**f8**	**f6**	**64**	**86**	**68**	**98**	**16**	**d4**	**a4**	**5c**	**cc**	**5d**	**65**	**b6**	**92**
	5	**6c**	**70**	**48**	**50**	**fd**	**ed**	**b9**	**da**	**5e**	**15**	**46**	**57**	**a7**	**8d**	**9d**	**84**
	6	**90**	**d8**	**ab**	**00**	**8c**	**bc**	**d3**	**0a**	**f7**	**e4**	**58**	**05**	**b8**	**b3**	**45**	**06**
	7	**d0**	**2c**	**1e**	**8f**	**ca**	**3f**	**0f**	**02**	**c1**	**af**	**bd**	**03**	**01**	**13**	**8a**	**6b**
	8	**3a**	**91**	**11**	**41**	**4f**	**67**	**dc**	**ea**	**97**	**f2**	**cf**	**ce**	**f0**	**b4**	**e6**	**73**
	9	**96**	**ac**	**74**	**22**	**e7**	**ad**	**35**	**85**	**e2**	**f9**	**37**	**e8**	**1c**	**75**	**df**	**6e**
	a	**47**	**f1**	**1a**	**71**	**1d**	**29**	**c5**	**89**	**6f**	**b7**	**62**	**0e**	**aa**	**18**	**be**	**1b**
	b	**fc**	**56**	**3e**	**4b**	**c6**	**d2**	**79**	**20**	**9a**	**db**	**c0**	**fe**	**78**	**cd**	**5a**	**f4**
	c	**1f**	**dd**	**a8**	**33**	**88**	**07**	**c7**	**31**	**b1**	**12**	**10**	**59**	**27**	**80**	**ec**	**5f**
	d	**60**	**51**	**7f**	**a9**	**19**	**b5**	**4a**	**0d**	**2d**	**e5**	**7a**	**9f**	**93**	**c9**	**9c**	**ef**
	e	**a0**	**e0**	**3b**	**4d**	**ae**	**2a**	**f5**	**b0**	**c8**	**eb**	**bb**	**3c**	**83**	**53**	**99**	**61**
	f	**17**	**2b**	**04**	**7e**	**ba**	**77**	**d6**	**26**	**e1**	**69**	**14**	**63**	**55**	**21**	**0c**	**7d**

5.3 SUMMARY OF TERMS

- **"Legacy SSE":** Refers to SSE, SSE2, SSE3, SSSE3, SSE4, and any future instruction sets referencing XMM registers and encoded without a VEX prefix.
- **XGETBV, XSETBV, XSAVE, XRSTOR** are defined in *IA-32 Intel Architecture Software Developer's Manual, Volumes 3A* and *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.
- **VEX:** refers to a two-byte or three-byte prefix. AVX and FMA instructions are encoded using a VEX prefix.
- **VEX.vvvv**. The VEX bitfield specifying a source or destination register (in 1's complement form).
- **rm_field**: shorthand for the ModR/M *r/m* field and any REX.B
- **reg_field:** shorthand for the ModR/M *reg* field and any REX.R

5.4 INSTRUCTION SET REFERENCE

<only instructions modified by AVX are included>

ADDPD - Add Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 58 /r ADDPD xmm1, xmm2/m128	A	V/V	SSE2	Add packed double-precision floating-point values from xmm2/mem to xmm1 and stores result in xmm1
VEX.NDS.128.66.0F.WIG 58 /r VADDPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Add packed double-precision floating-point values from xmm3/mem to xmm2 and stores result in xmm1
VEX.NDS.256.66.0F.WIG 58 /r VADDPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add packed double-precision floating-point values from ymm3/mem to ymm2 and stores result in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD add of the two or four packed double-precision floating-point values from the first Source operand to the Second Source operand, and stores the packed double-precision floating-point results in the destination operand.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VADDPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
DEST[127:64] ← SRC1[127:64] + SRC2[127:64]
DEST[191:128] ← SRC1[191:128] + SRC2[191:128]
DEST[255:192] ← SRC1[255:192] + SRC2[255:192]

.

VADDPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
DEST[127:64] ← SRC1[127:64] + SRC2[127:64]
DEST[255:128] ← 0

ADDPD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] + SRC[63:0]
DEST[127:64] ← DEST[127:64] + SRC[127:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VADDPD __m256d _mm256_add_pd (__m256d a, __m256d b);

ADDPD __m128d _mm_add_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

ADDPS- Add Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 58 /r ADDPS xmm1, xmm2/m128	A	V/V	SSE	Add packed single-precision floating-point values from xmm2/mem to xmm1 and stores result in xmm1
VEX.NDS.128.0F.WIG 58 /r VADDPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Add packed single-precision floating-point values from xmm3/mem to xmm2 and stores result in xmm1
VEX.NDS.256.0F.WIG 58 /r VADDPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add packed single-precision floating-point values from ymm3/mem to ymm2 and stores result in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD add of the four or eight packed single-precision floating-point values from the first Source operand to the Second Source operand, and stores the packed single-precision floating-point results in the destination operand.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VADDPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] + SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[159:128] ← SRC1[159:128] + SRC2[159:128]
DEST[191:160]← SRC1[191:160] + SRC2[191:160]
DEST[223:192] ← SRC1[223:192] + SRC2[223:192]
DEST[255:224] ← SRC1[255:224] + SRC2[255:224].

VADDPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] + SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[255:128] ← 0

ADDPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] + SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VADDPS __m256 _mm256_add_ps (__m256 a, __m256 b);

ADDPS __m128 _mm_add_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

ADDSD- Add Scalar Double Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 58 /r ADDSD xmm1, xmm2/m64	A	V/V	SSE2	Add the low double-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1
VEX.NDS.LIG.F2.0F.WIG 58 /r VADDSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Add the low double-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

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

128-bit Legacy SSE version: Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

Operation

VADDSD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0

ADDSD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] + SRC[63:0]

DEST[255:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

ADDSD __m128d _mm_add_sd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

ADDSS- Add Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 58 /r ADDSS xmm1, xmm2/m32	A	V/V	SSE	Add the low single-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1
VEX.NDS.LIG.F3.0F.WIG 58 /r VADDSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Add the low single-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

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

128-bit Legacy SSE version: Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

Operation

VADDSS DEST, SRC1, SRC2 (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[255:128] ← 0

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

DEST[31:0] ← DEST[31:0] + SRC[31:0]
DEST[255:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

ADDSS __m128 _mm_add_ss (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

ADDSUBPD- Packed Double FP Add/Subtract

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F D0 /r ADDSUBPD xmm1, xmm2/m128	A	V/V	SSE3	Add/subtract double-precision floating-point values from xmm2/m128 to xmm1
VEX.NDS.128.66.0F.WIG D0 /r VADDSUBPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add/subtract packed double-precision floating-point values from xmm3/mem to xmm2 and stores result in xmm1
VEX.NDS.256.66.0F.WIG D0 /r VADDSUBPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add / subtract packed double-precision floating-point values from ymm3/mem to ymm2 and stores result in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Adds odd-numbered double-precision floating-point values of the first source operand (second operand) with the corresponding double-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered double-precision floating-point values from the second source operand from the corresponding double-precision floating values in the first source operand; stores the result into the even-numbered values of the destination operand

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VADDSUBPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
DEST[127:64] ← SRC1[127:64] + SRC2[127:64]
DEST[191:128] ← SRC1[191:128] - SRC2[191:128]
DEST[255:192] ← SRC1[255:192] + SRC2[255:192]

VADDSUBPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
DEST[127:64] ← SRC1[127:64] + SRC2[127:64]
DEST[255:128] ← 0

ADDSUBPD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] - SRC[63:0]
DEST[127:64] ← DEST[127:64] + SRC[127:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VADDSUBPD __m256d _mm256_addsub_pd (__m256d a, __m256d b);

ADDSUBPD __m128d _mm_addsub_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

ADDSUBPS- Packed Single FP Add/Subtract

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F D0 /r ADDSUBPS xmm1, xmm2/m128	A	V/V	SSE3	Add/subtract single-precision floating-point values from xmm2/m128 to xmm1
VEX.NDS.128.F2.0F.WIG D0 /r VADDSUBPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add/subtract single-precision floating-point values from xmm3/mem to xmm2 and stores result in xmm1
VEX.NDS.256.F2.0F.WIG D0 /r VADDSUBPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add / subtract single-precision floating-point values from ymm3/mem to ymm2 and stores result in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Adds odd-numbered single-precision floating-point values of the first source operand (second operand) with the corresponding single-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered single-precision floating-point values from the second source operand from the corresponding single-precision floating values in the first source operand; stores the result into the even-numbered values of the destination operand.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register

and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VADDSUBPS (VEX.256 encoded version)

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

VADDSUBPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[255:128] ← 0

ADDSUBPS (128-bit Legacy SSE version)

DEST[31:0] ← DEST[31:0] - SRC[31:0]
DEST[63:32] ← DEST[63:32] + SRC[63:32]
DEST[95:64] ← DEST[95:64] - SRC[95:64]
DEST[127:96] ← DEST[127:96] + SRC[127:96]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VADDSUBPS __m256 _mm256_addsub_ps (__m256 a, __m256 b);

ADDSUBPS __m128 _mm_addsub_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

AESENC/AESENCLAST- Perform One Round of an AES Encryption Flow

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 DC /r AESENC xmm1, xmm2/m128	A	V/V	AES	Perform one round of an AES encryption flow, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
66 0F 38 DD /r AESENCLAST xmm1, xmm2/m128	A	V/V	AES	Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DC /r VAESENC xmm1, xmm2, xmm3/m128	B	V/V	Both AES and AVX flags	Perform one round of an AES encryption flow, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from the xmm3/m128; store the result in xmm1.
VEX.NDS.128.66.0F38.WIG DD /r VAESENCLAST xmm1, xmm2, xmm3/m128	B	V/V	Both AES and AVX flags	Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from xmm2 with a 128 bit round key from xmm3/m128; store the result in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

These instructions perform a single round of an AES encryption flow using a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.

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

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (255:128) of the destination YMM register are zeroed.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (255:128) of the corresponding YMM destination register remain unchanged.

Operation

VAESENC

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
STATE ← MixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[255:128] ← 0
```

AESENC

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
STATE ← MixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[255:128] (Unmodified)
```

VAESENCLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[255:128] ← 0
```

AESENCLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
```

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESENC __m128i _mm_aesenc (__m128i, __m128i)

(V)AESENCLAST __m128i _mm_aesenclast (__m128i, __m128i)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4

AESDEC/AESDECLAST- Perform One Round of an AES Decryption Flow

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 DE /r AESDEC xmm1, xmm2/m128	A	V/V	AES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
66 0F 38 DF /r AESDECLAST xmm1, xmm2/m128	A	V/V	AES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128	B	V/V	Both AES and AVX flags	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from xmm3/m128; store the result in xmm1.
VEX.NDS.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128	B	V/V	Both AES and AVX flags	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from xmm3/m128; store the result in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

These instructions perform a single round of the AES decryption flow using the Equivalent Inverse Cipher, with the round key from the second source operand, operating on a 128-bit data (state) from the first source operand, and store the result in the destination operand.

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

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (255:128) of the destination YMM register are zeroed.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (255:128) of the corresponding YMM destination register remain unchanged.

Operation

VAESDEC

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
STATE ← InvMixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[255:128] ← 0
```

AESDEC

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
STATE ← InvMixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[255:128] (Unmodified)
```

VAESDECLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
```

STATE ← InvShiftRows(STATE);
STATE ← InvSubBytes(STATE);
DEST[127:0] ← STATE XOR RoundKey;
DEST[255:128] ← 0

AESDECLAST
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows(STATE);
STATE ← InvSubBytes(STATE);
DEST[127:0] ← STATE XOR RoundKey;
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESDEC __m128i _mm_aesdec (__m128i, __m128i)

(V)AESDECLAST __m128i _mm_aesdeclast (__m128i, __m128i)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4

AESIMC- Perform the AES InvMixColumn Transformation

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 DB /r AESIMC xmm1, xmm2/m128	A	V/V	AES	Perform the InvMixColumn transformation on a 128-bit round key from xmm2/m128 and store the result in xmm1
VEX.128.66.0F38.WIG DB /r VAESIMC xmm1, xmm2/m128	A	V/V	Both AES and AVX flags	Perform the InvMixColumn transformation on a 128-bit round key from xmm2/m128 and store the result in xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Perform the InvMixColumns transformation on the source operand and store the result in the destination operand. The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location.

Note the AESIMC instruction should be applied to the expanded AES round keys (except for the first and last round key) in order to prepare them for decryption using the "Equivalent Inverse Cipher" (defined in FIPS 197).

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

Operation

VAESIMC

DEST[127:0] ← InvMixColumns(SRC);
DEST[255:128] ← 0;

AESIMC

DEST[127:0] ← InvMixColumns(SRC);
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESIMC __m128i _mm_aesimc (__m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

AESKEYGENASSIST - AES Round Key Generation Assist

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A DF /r ib AESKEYGENASSIST xmm1, xmm2/m128, imm8	A	V/V	AES	Assist in AES round key generation using an 8 bits Round Constant (RCON) specified in the immediate byte, operating on 128 bits of data specified in xmm2/m128 and stores the result in xmm1.
VEX.128.66.0F3A.WIG DF /r ib VAESKEYGENASSIST xmm1, xmm2/m128, imm8	A	V/V	Both AES and AVX flags	Assist in AES round key generation using 8 bits Round Constant (RCON) specified in the immediate byte, operating on 128 bits of data specified in xmm2/m128 and stores the result in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Assist in expanding the AES cipher key, by computing steps towards generating a round key for encryption, using 128-bit data specified in the source operand and an 8-bit round constant specified as an immediate, store the result in the destination operand.

The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

128-bit Legacy SSE version:Bits (255:128) of the corresponding YMM destination register remain unchanged.

Operation

VAESKEYGENASSIST

X3[31:0] ← SRC [127: 96];
X2[31:0] ← SRC [95: 64];
X1[31:0] ← SRC [63: 32];
X0[31:0] ← SRC [31: 0];

RCON[31:0] ← ZeroExtend(Imm8[7:0]);
DEST[31:0] ← SubWord(X1);
DEST[63:32] ← RotWord(SubWord(X1)) XOR RCON;
DEST[95:64] ← SubWord(X3);
DEST[127:96] ← RotWord(SubWord(X3)) XOR RCON;
DEST[255:128] ← 0;

AESKEYGENASSIST
X3[31:0] ← SRC [127: 96];
X2[31:0] ← SRC [95: 64];
X1[31:0] ← SRC [63: 32];
X0[31:0] ← SRC [31: 0];
RCON[31:0] ← ZeroExtend(Imm8[7:0]);
DEST[31:0] ← SubWord(X1);
DEST[63:32] ← RotWord(SubWord(X1)) XOR RCON;
DEST[95:64] ← SubWord(X3);
DEST[127:96] ← RotWord(SubWord(X3)) XOR RCON;
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESKEYGENASSIST __m128i _mm_aesimc (__m128i, const int)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

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

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 54 /r ANDPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical AND of packed double-precision floating-point values in xmm1 and xmm2/mem
VEX.NDS.128.66.0F.WIG 54 /r VANDPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND of packed double-precision floating-point values in xmm2 and xmm3/mem
VEX.NDS.256.66.0F.WIG 54 /r VANDPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND of packed double-precision floating-point values in ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND of the two or four packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VANDPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE AND SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE AND SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE AND SRC2[255:192]

VANDPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE AND SRC2[127:64]
DEST[255:128] ← 0

ANDPD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] BITWISE AND SRC[63:0]
DEST[127:64] ← DEST[127:64] BITWISE AND SRC[127:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VANDPD __m256d _mm256_and_pd (__m256d a, __m256d b);

ANDPD __m128d _mm_and_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

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

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 54 /r ANDPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical AND of packed single-precision floating-point values in xmm1 and xmm2/mem
VEX.NDS.128.0F.WIG 54 /r VANDPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/mem
VEX.NDS.256.0F.WIG 54 /r VANDPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND of the four or eight packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VANDPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE AND SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE AND SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE AND SRC2[127:96]
DEST[159:128] ← SRC1[159:128] BITWISE AND SRC2[159:128]
DEST[191:160]← SRC1[191:160] BITWISE AND SRC2[191:160]
DEST[223:192] ← SRC1[223:192] BITWISE AND SRC2[223:192]
DEST[255:224] ← SRC1[255:224] BITWISE AND SRC2[255:224].

VANDPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE AND SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE AND SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE AND SRC2[127:96]
DEST[255:128] ← 0

ANDPS (128-bit Legacy SSE version)

DEST[31:0] ← DEST[31:0] BITWISE AND SRC[31:0]
DEST[63:32] ← DEST[63:32] BITWISE AND SRC[63:32]
DEST[95:64] ← DEST[95:64] BITWISE AND SRC[95:64]
DEST[127:96] ← DEST[127:96] BITWISE AND SRC[127:96]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VANDPS __m256 _mm256_and_ps (__m256 a, __m256 b);

ANDPS __m128 _mm_and_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

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

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 55 /r ANDNPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical AND NOT of packed double-precision floating-point values in xmm1 and xmm2/mem
VEX.NDS.128.66.0F.WIG 55 /r VANDNPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND NOT of packed double-precision floating-point values in xmm2 and xmm3/mem
VEX.NDS.256.66.0F.WIG 55/r VANDNPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND NOT of packed double-precision floating-point values in ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND NOT of the two or four packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VANDNPD (VEX.256 encoded version)

DEST[63:0] ← (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]
DEST[127:64] ← (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]
DEST[191:128] ← (NOT(SRC1[191:128])) BITWISE AND SRC2[191:128]
DEST[255:192] ← (NOT(SRC1[255:192])) BITWISE AND SRC2[255:192]

VANDNPD (VEX.128 encoded version)

DEST[63:0] ← (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]
DEST[127:64] ← (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]
DEST[255:128] ← 0

ANDNPD (128-bit Legacy SSE version)

DEST[63:0] ← (NOT(DEST[63:0])) BITWISE AND SRC[63:0]
DEST[127:64] ← (NOT(DEST[127:64])) BITWISE AND SRC[127:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VANDNPD __m256d _mm256_andnot_pd (__m256d a, __m256d b);

ANDNPD __m128d _mm_andnot_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

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

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 55 /r ANDNPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm1 and xmm2/mem
VEX.NDS.128.0F.WIG 55 /r VANDNPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm2 and xmm3/mem
VEX.NDS.256.0F.WIG 55 /r VANDNPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND NOT of the four or eight packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VANDNPS (VEX.256 encoded version)

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

VANDNPS (VEX.128 encoded version)

DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[63:32] ← (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]
DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[255:128] ← 0

ANDNPS (128-bit Legacy SSE version)

DEST[31:0] ← (NOT(DEST[31:0])) BITWISE AND SRC[31:0]
DEST[63:32] ← (NOT(DEST[63:32])) BITWISE AND SRC[63:32]
DEST[95:64] ← (NOT(DEST[95:64])) BITWISE AND SRC[95:64]
DEST[127:96] ← (NOT(DEST[127:96])) BITWISE AND SRC[127:96]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VANDNPS __m256 _mm256_andnot_ps (__m256 a, __m256 b);

ANDNPS __m128 _mm_andnot_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

BLENDPD- Blend Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0D /r ib BLENDPD xmm1, xmm2/m128, imm8	A	V/V	SSE4_1	Select packed double-precision floating-point Values from xmm1 and xmm2/m128 from mask in imm8 and store the values in xmm1
VEX.NDS.128.66.0F3A.WIG 0D /r ib VBLENDPD xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Select packed double-precision floating-point Values from xmm2 and xmm3/m128 from mask in imm8 and store the values in xmm1
VEX.NDS.256.66.0F3A.WIG 0D /r ib VBLENDPD ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Select packed double-precision floating-point Values from ymm2 and ymm3/m256 from mask in imm8 and store the values in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[3:0]

Description

Double-precision floating-point values from the second source operand (third operand) are conditionally merged with values from the first source operand (second operand) and written to the destination operand (first operand). The immediate bits [3:0] determine whether the corresponding double-precision floating-point value in the destination is copied from the second source or first source. If a bit in the mask, corresponding to a word, is "1", then the double-precision floating-point value in the second source operand is copied, else the value in the first source operand is copied.

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

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

operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VBLENDPD (VEX.256 encoded version)

```
IF (IMM8[0] = 0)THEN DEST[63:0] ← SRC1[63:0]
        ELSE DEST [63:0] ← SRC2[63:0] FI
IF (IMM8[1] = 0) THEN DEST[127:64] ← SRC1[127:64]
        ELSE DEST [127:64] ← SRC2[127:64] FI
IF (IMM8[2] = 0) THEN DEST[191:128] ← SRC1[191:128]
        ELSE DEST [191:128] ← SRC2[191:128] FI
IF (IMM8[3] = 0) THEN DEST[255:192] ← SRC1[255:192]
        ELSE DEST [255:192] ← SRC2[255:192] FI
```

VBLENDPD (VEX.128 encoded version)

```
IF (IMM8[0] = 0)THEN DEST[63:0] ← SRC1[63:0]
        ELSE DEST [63:0] ← SRC2[63:0] FI
IF (IMM8[1] = 0) THEN DEST[127:64] ← SRC1[127:64]
        ELSE DEST [127:64] ← SRC2[127:64] FI
DEST[255:128] ← 0
```

BLENDPD (128-bit Legacy SSE version)

```
IF (IMM8[0] = 0)THEN DEST[63:0] ← DEST[63:0]
        ELSE DEST [63:0] ← SRC[63:0] FI
IF (IMM8[1] = 0) THEN DEST[127:64] ← DEST[127:64]
        ELSE DEST [127:64] ← SRC[127:64] FI
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VBLENDPD __m256d _mm256_blend_pd (__m256d a, __m256d b, const int mask);

BLENDPD __m128d _mm_blend_pd (__m128d a, __m128d b, const int mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

BLENDPS- Blend Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0C /r ib BLENDPS xmm1, xmm2/m128, imm8	A	V/V	SSE4_1	Select packed single-precision floating-point values from xmm1 and xmm2/m128 from mask in imm8 and store the values in xmm1
VEX.NDS.128.66.0F3A.WIG 0C /r ib VBLENDPS xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Select packed single-precision floating-point values from xmm2 and xmm3/m128 from mask in imm8 and store the values in xmm1
VEX.NDS.256.66.0F3A.WIG 0C /r ib VBLENDPS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Select packed single-precision floating-point values from ymm2 and ymm3/m256 from mask in imm8 and store the values in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[3:0]

Description

Single-precision floating-point values from the second source operand (third operand) are conditionally merged with values from the first source operand (second operand) and written to the destination operand (first operand). The immediate bits [7:0] determine whether the corresponding single precision floating-point value in the destination is copied from the second source or first source. If a bit in the mask, corresponding to a word, is "1", then the single-precision floating-point value in the second source operand is copied, else the value in the first source operand is copied.

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

VEX.128 encoded version: The first source operand an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VBLENDPS (VEX.256 encoded version)

```
IF (IMM8[0] = 0) THEN DEST[31:0] ←SRC1[31:0]
        ELSE DEST [31:0] ← SRC2[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] ← SRC1[63:32]
        ELSE DEST [63:32] ← SRC2[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← SRC1[95:64]
        ELSE DEST [95:64] ← SRC2[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] ← SRC1[127:96]
        ELSE DEST [127:96] ← SRC2[127:96] FI
IF (IMM8[4] = 0) THEN DEST[159:128] ← SRC1[159:128]
        ELSE DEST [159:128] ← SRC2[159:128] FI
IF (IMM8[5] = 0) THEN DEST[191:160] ← SRC1[191:160]
        ELSE DEST [191:160] ← SRC2[191:160] FI
IF (IMM8[6] = 0) THEN DEST[223:192] ← SRC1[223:192]
        ELSE DEST [223:192] ← SRC2[223:192] FI
IF (IMM8[7] = 0) THEN DEST[255:224] ← SRC1[255:224]
        ELSE DEST [255:224] ← SRC2[255:224] FI.
```

VBLENDPS (VEX.128 encoded version)

```
IF (IMM8[0] = 0) THEN DEST[31:0] ←SRC1[31:0]
        ELSE DEST [31:0] ← SRC2[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] ← SRC1[63:32]
        ELSE DEST [63:32] ← SRC2[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← SRC1[95:64]
        ELSE DEST [95:64] ← SRC2[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] ← SRC1[127:96]
        ELSE DEST [127:96] ← SRC2[127:96] FI
DEST[255:128] ← 0
```

BLENDPS (128-bit Legacy SSE version)

```
IF (IMM8[0] = 0) THEN DEST[31:0] ←DEST[31:0]
        ELSE DEST [31:0] ← SRC[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] ← DEST[63:32]
```

```
        ELSE DEST [63:32] ← SRC[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← DEST[95:64]
        ELSE DEST [95:64] ← SRC[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] ← DEST[127:96]
        ELSE DEST [127:96] ← SRC[127:96] FI
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VBLENDPS __m256 _mm256_blend_ps (__m256 a, __m256 b, const int mask);

BLENDPS __m128 _mm_blend_ps (__m128 a, __m128 b, const int mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

BLENDVPD- Blend Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 15 /r BLENDVPD xmm1, xmm2/m128, <XMM0>	A	V/V	SSE4_1	Conditionally copy double-precision floating-point values from xmm1 or xmm2/m128 to xmm1, based on mask bits in the implicit mask operand, XMM0.
VEX.NDS.128.66.0F3A.W0 4B /r /is4 VBLENDVPD xmm1, xmm2, xmm3/m128, xmm4	B	V/V	AVX	Conditionally copy double-precision floating-point values from xmm2 or xmm3/m128 to xmm1, based on mask bits in the mask operand, xmm4
VEX.NDS.256.66.0F3A.W0 4B /r /is4 VBLENDVPD ymm1, ymm2, ymm3/m256, ymm4	B	V/V	AVX	Conditionally copy double-precision floating-point values from ymm2 or ymm3/m256 to ymm1, based on mask bits in the mask operand, ymm4

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

Description

Conditionally copy each quadword data element of double-precision floating-point value from the second source operand and the first source operand depending on mask bits defined in the mask register operand. The mask bits are the most significant bit in each quadword element of the mask register.

Each quadword element of the destination operand is copied from:

- the corresponding quadword element in the second source operand, If a mask bit is "1"; or
- the corresponding quadword element in the first source operand, If a mask bit is "0"

The register assignment of the implicit mask operand for BLENDVPD is defined to be the architectural register XMM0

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (255:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPD with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (255:128) of the corresponding YMM register (destination register) are zeroed. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and destination operand are YMM registers. The second source operand can be a YMM register or a 256-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. VEX.W must be 0, otherwise, the instruction will #UD.

VBLENDVPD permits the mask to be any XMM or YMM register. In contrast, BLENDVPD treats XMM0 implicitly as the mask and do not support non-destructive destination operation.

Operation

VBLENDVPD (VEX.256 encoded version)

```
MASK ← SRC3
IF (MASK[63] = 0) THEN DEST[63:0] ← SRC1[63:0]
        ELSE DEST [63:0] ← SRC2[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] ← SRC1[127:64]
        ELSE DEST [127:64] ← SRC2[127:64] FI
IF (MASK[191] = 0) THEN DEST[191:128] ← SRC1[191:128]
        ELSE DEST [191:128] ← SRC2[191:128] FI
IF (MASK[255] = 0) THEN DEST[255:192] ← SRC1[255:192]
        ELSE DEST [255:192] ← SRC2[255:192] FI
```

VBLENDVPD (VEX.128 encoded version)

```
MASK ← SRC3
IF (MASK[63] = 0) THEN DEST[63:0] ← SRC1[63:0]
        ELSE DEST [63:0] ← SRC2[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] ← SRC1[127:64]
        ELSE DEST [127:64] ← SRC2[127:64] FI
DEST[255:128] ← 0
```

BLENDVPD (128-bit Legacy SSE version)

```
MASK ← XMM0
IF (MASK[63] = 0) THEN DEST[63:0] ← DEST[63:0]
        ELSE DEST [63:0] ← SRC[63:0] FI
```

```
IF (MASK[127] = 0) THEN DEST[127:64] ← DEST[127:64]
        ELSE DEST [127:64] ← SRC[127:64] FI
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VBLENDVPD __m256 _mm256_blendv_pd (__m256d a, __m256d b, __m256d mask);

VBLENDVPD __m128 _mm_blendv_pd (__m128d a, __m128d b, __m128d mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.W = 1.

BLENDVPS- Blend Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 14 /r BLENDVPS xmm1, xmm2/m128, <XMM0>	A	V/V	SSE4_1	Conditionally copy single-precision floating-point values from xmm2 or xmm3/m128 to xmm1, based on mask bits in the implicit mask operand, XMM0.
VEX.NDS.128.66.0F3A.W0 4A /r /is4 VBLENDVPS xmm1, xmm2, xmm3/m128, xmm4	B	V/V	AVX	Conditionally copy single-precision floating-point values from xmm2 or xmm3/m128 to xmm1, based on mask bits in the specified mask operand, xmm4
VEX.NDS.256.66.0F3A.W0 4A /r /is4 VBLENDVPS ymm1, ymm2, ymm3/m256, ymm4	B	V/V	AVX	Conditionally copy single-precision floating-point values from ymm2 or ymm3/m256 to ymm1, based on mask bits in the specified mask register, ymm4

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

Description

Conditionally copy each dword data element of single-precision floating-point value from the second source operand and the first source operand depending on mask bits defined in the mask register operand. The mask bits are the most significant bit in each dword element of the mask register.

Each quadword element of the destination operand is copied from:

- the corresponding dword element in the second source operand, If a mask bit is "1"; or
- the corresponding dword element in the first source operand, If a mask bit is "0"

The register assignment of the implicit mask operand for BLENDVPS is defined to be the architectural register XMM0

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (255:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPS with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (255:128) of the corresponding YMM register (destination register) are zeroed. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and destination operand are YMM registers. The second source operand can be a YMM register or a 256-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. VEX.W must be 0, otherwise, the instruction will #UD.

VBLENDVPS permits the mask to be any XMM or YMM register. In contrast, BLENDVPS treats XMM0 implicitly as the mask and do not support non-destructive destination operation.

Operation

VBLENDVPS (VEX.256 encoded version)

```
MASK ← SRC3
IF (MASK[31] = 0) THEN DEST[31:0] ← SRC1[31:0]
        ELSE DEST [31:0] ← SRC2[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← SRC1[63:32]
        ELSE DEST [63:32] ← SRC2[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] ← SRC1[95:64]
        ELSE DEST [95:64] ← SRC2[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← SRC1[127:96]
        ELSE DEST [127:96] ← SRC2[127:96] FI
IF (MASK[159] = 0) THEN DEST[159:128] ← SRC1[159:128]
        ELSE DEST [159:128] ← SRC2[159:128] FI
IF (MASK[191] = 0) THEN DEST[191:160] ← SRC1[191:160]
        ELSE DEST [191:160] ← SRC2[191:160] FI
IF (MASK[223] = 0) THEN DEST[223:192] ← SRC1[223:192]
        ELSE DEST [223:192] ← SRC2[223:192] FI
IF (MASK[255] = 0) THEN DEST[255:224] ← SRC1[255:224]
        ELSE DEST [255:224] ← SRC2[255:224] FI
```

VBLENDVPS (VEX.128 encoded version)

```
MASK ← SRC3
IF (MASK[31] = 0) THEN DEST[31:0] ← SRC1[31:0]
```

```
        ELSE DEST [31:0] ← SRC2[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← SRC1[63:32]
        ELSE DEST [63:32] ← SRC2[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] ← SRC1[95:64]
        ELSE DEST [95:64] ← SRC2[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← SRC1[127:96]
        ELSE DEST [127:96] ← SRC2[127:96] FI
DEST[255:128] ← 0
```

BLENDVPS (128-bit Legacy SSE version)

```
MASK ← XMM0
IF (MASK[31] = 0) THEN DEST[31:0] ← DEST[31:0]
        ELSE DEST [31:0] ← SRC[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← DEST[63:32]
        ELSE DEST [63:32] ← SRC[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] ← DEST[95:64]
        ELSE DEST [95:64] ← SRC[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← DEST[127:96]
        ELSE DEST [127:96] ← SRC[127:96] FI
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VBLENDVPS __m256 _mm256_blendv_ps (__m256 a, __m256 b, __m256 mask);

VBLENDVPS __m128 _mm_blendv_ps (__m128 a, __m128 b, __m128 mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.W = 1.

VBROADCAST- Load with Broadcast

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1, m32	A	V/V	AVX	Broadcast single-precision floating-point element in mem to four locations in xmm1
VEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1, m32	A	V/V	AVX	Broadcast single-precision floating-point element in mem to eight locations in ymm1
VEX.256.66.0F38.W0 19 /r VBROADCASTSD ymm1, m64	A	V/V	AVX	Broadcast double-precision floating-point element in mem to four locations in ymm1
VEX.256.66.0F38.W0 1A /r VBROADCASTF128 ymm1, m128	A	V/V	AVX	Broadcast 128 bits of floating-point data in mem to low and high 128-bits in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Load floating point values from the source operand (second operand) and broadcast to all elements of the destination operand (first operand).

The destination operand is a YMM register. The source operand is either a 32-bit, 64-bit, or 128-bit memory location. Register source encodings are reserved and will #UD.

VBROADCASTSD and VBROADCASTF128 are only supported as 256-bit wide versions. VBROADCASTSS is supported in both 128-bit and 256-bit wide versions.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

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

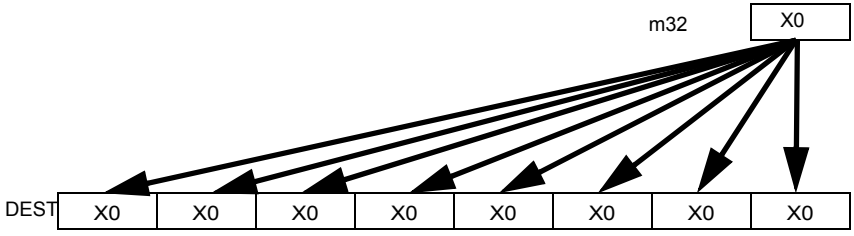


Figure 5-1. VBROADCASTSS Operation (VEX.256 encoded version)

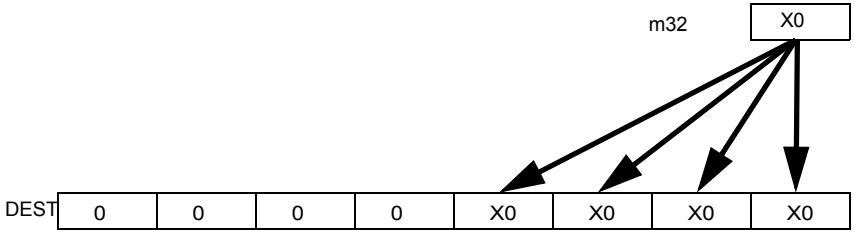


Figure 5-2. VBROADCASTSS Operation (128-bit version)

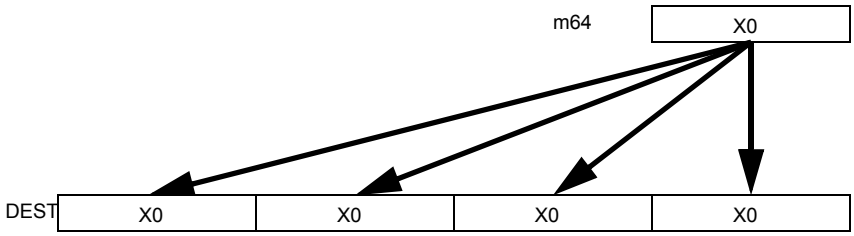


Figure 5-3. VBROADCASTSD Operation

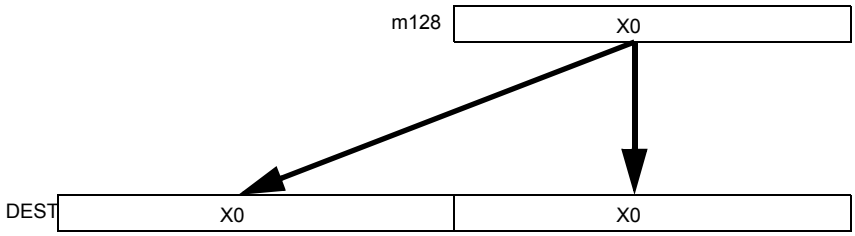


Figure 5-4. VBROADCASTF128 Operation

Operation

VBROADCASTSS (128 bit version)
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[255:128] ← 0

VBROADCASTSS (VEX.256 encoded version)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[159:128] ← temp
DEST[191:160] ← temp
DEST[223:192] ← temp
DEST[255:224] ← temp
```

VBROADCASTSD (VEX.256 encoded version)

```
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp
```

VBROADCASTF128

```
temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[255:128] ← temp
```

Intel C/C++ Compiler Intrinsic Equivalent

VBROADCASTSS __m128 _mm_broadcast_ss(float *a);

VBROADCASTSS __m256 _mm256_broadcast_ss(float *a);

VBROADCASTSD __m256d _mm256_broadcast_sd(double *a);

VBROADCASTF128 __m256 _mm256_broadcast_ps(__m128 * a);

VBROADCASTF128 __m256d _mm256_broadcast_pd(__m128d * a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0 for VBROADCASTSD
	If VEX.L = 0 for VBROADCASTF128
	If VEX.W = 1

CMPPD- Compare Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C2 /r ib CMPPD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Compare packed double-precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate
VEX.NDS.128.66.0F.WIG C2 /r ib VCMPPD xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Compare packed double-precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate
VEX.NDS.256.66.0F.WIG C2 /r ib VCMPPD ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Compare packed double-precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed double-precision floating-point values in the second source operand and the first source operand and returns the results of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Four comparisons are performed with results written to the destination operand.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM

register or 128-bit memory location. Bits (255:128) of the corresponding YMM destination register remain unchanged. Two comparisons are performed with results written to bits 127:0 of the destination operand.

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (255:128) of the destination YMM register are zeroed. Two comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 5-9). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 5-9). Bits 3 through 7 of the immediate are reserved.

Table 5-9. Comparison Predicate for CMPPD and CMPPS Instructions

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNAN
			A >B	A < B	A = B	Unordered[1]	
EQ_OQ (EQ)	0H	Equal (ordered, non-signaling)	False	False	True	False	No
LT_OS (LT)	1H	Less-than (ordered, signaling)	False	True	False	False	Yes
LE_OS (LE)	2H	Less-than-or-equal (ordered, signaling)	False	True	True	False	Yes
UNORD_Q (UNORD)	3H	Unordered (non-signaling)	False	False	False	True	No
NEQ_UQ (NEQ)	4H	Not-equal (unordered, non-signaling)	True	True	False	True	No
NLT_US (NLT)	5H	Not-less-than (unordered, signaling)	True	False	True	True	Yes
NLE_US (NLE)	6H	Not-less-than-or-equal (unordered, signaling)	True	False	False	True	Yes
ORD_Q (ORD)	7H	Ordered (non-signaling)	True	True	True	False	No

Table 5-9. Comparison Predicate for CMPPD and CMPPS Instructions (Continued)

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNAN
			A >B	A < B	A = B	Unordered[1]	
EQ_UQ	8H	Equal (unordered, non-signaling)	False	False	True	True	No
NGE_US (NGE)	9H	Not-greater-than-or-equal (unordered, signaling)	False	True	False	True	Yes
NGT_US (NGT)	AH	Not-greater-than (unordered, signaling)	False	True	True	True	Yes
FALSE_OQ(FALSE)	BH	False (ordered, non-signaling)	False	False	False	False	No
NEQ_OQ	CH	Not-equal (ordered, non-signaling)	True	True	False	False	No
GE_OS (GE)	DH	Greater-than-or-equal (ordered, signaling)	True	False	True	False	Yes
GT_OS (GT)	EH	Greater-than (ordered, signaling)	True	False	False	False	Yes
TRUE_UQ(TRUE)	FH	True (unordered, non-signaling)	True	True	True	True	No
EQ_OS	10H	Equal (ordered, signaling)	False	False	True	False	Yes
LT_OQ	11H	Less-than (ordered, nonsignaling)	False	True	False	False	No
LE_OQ	12H	Less-than-or-equal (ordered, nonsignaling)	False	True	True	False	No
UNORD_S	13H	Unordered (signaling)	False	False	False	True	Yes
NEQ_US	14H	Not-equal (unordered, signaling)	True	True	False	True	Yes
NLT_UQ	15H	Not-less-than (unordered, nonsignaling)	True	False	True	True	No
NLE_UQ	16H	Not-less-than-or-equal (unordered, nonsignaling)	True	False	False	True	No

Table 5-9. Comparison Predicate for CMPPD and CMPPS Instructions (Continued)

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNAN
			A >B	A < B	A = B	Unordered[1]	
ORD_S	17H	Ordered (signaling)	True	True	True	False	Yes
EQ_US	18H	Equal (unordered, signaling)	False	False	True	True	Yes
NGE_UQ	19H	Not-greater-than-or-equal (unordered, nonsignaling)	False	True	False	True	No
NGT_UQ	1AH	Not-greater-than (unordered, nonsignaling)	False	True	True	True	No
FALSE_OS	1BH	False (ordered, signaling)	False	False	False	False	Yes
NEQ_OS	1CH	Not-equal (ordered, signaling)	True	True	False	False	Yes
GE_OQ	1DH	Greater-than-or-equal (ordered, nonsignaling)	True	False	True	False	No
GT_OQ	1EH	Greater-than (ordered, nonsignaling)	True	False	False	False	No
TRUE_US	1FH	True (unordered, signaling)	True	True	True	True	Yes

NOTES:

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

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

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

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the

program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 5-10. Compiler should treat reserved Imm8 values as illegal syntax.

Table 5-10. Pseudo-Op and CMPPD Implementation

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

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

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 5-9, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPD instruction. See Table 5-11, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

Table 5-11. Pseudo-Op and VCMPPD Implementation

Pseudo-Op	CMPPD Implementation
VCMPEQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0*
VCMPLTPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1*
VCMPLEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 2*
VCMPUNORDPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 3*
VCMPNEQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 4*

Table 5-11. Pseudo-Op and VCMPPD Implementation

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

Operation

CASE (COMPARISON PREDICATE) OF

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

VCMPPD (VEX.256 encoded version)

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

```
IF CMP1 = TRUE
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0000000000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[191:128] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[191:128] ← 0000000000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[255:192] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[255:192] ← 0000000000000000H; FI;
```

VCMPPD (VEX.128 encoded version)

```
CMP0 ← SRC1[63:0] OP5 SRC2[63:0];
CMP1 ← SRC1[127:64] OP5 SRC2[127:64];
IF CMP0 = TRUE
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0000000000000000H; FI;
DEST[255:128] ← 0
```

CMPPD (128-bit Legacy SSE version)

```
CMP0 ← SRC1[63:0] OP3 SRC2[63:0];
CMP1 ← SRC1[127:64] OP3 SRC2[127:64];
IF CMP0 = TRUE
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0000000000000000H; FI;
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCMPPD __m256 _mm256_cmp_pd(__m256 a, __m256 b, const int imm)

VCMPPD __m128 _mm_cmp_pd(__m128 a, __m128 b, const int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 5-9.

Denormal

Other Exceptions

See Exceptions Type 2

CMPPS- Compare Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C2 /r ib CMPPS xmm1, xmm2/m128, imm8	A	V/V	SSE	Compare packed single-precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEX.NDS.128.0F.WIG C2 /r ib VCMPPS xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Compare packed single-precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEX.NDS.256.0F.WIG C2 /r ib VCMPPS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Compare packed single-precision floating-point values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed single-precision floating-point values in the second source operand and the first source operand and returns the results of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each of the pairs of packed values. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Eight comparisons are performed with results written to the destination operand.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM

register or 128-bit memory location. Bits (255:128) of the corresponding YMM destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand.

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (255:128) of the destination YMM register are zeroed. Four comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Figure 5-9). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 5-9). Bits 3 through 7 of the immediate are reserved.

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

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

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

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 5-12. Compiler should treat reserved Imm8 values as illegal syntax.

Table 5-12. Pseudo-Op and CMPPS Implementation

Pseudo-Op	CMPPS Implementation
CMPEQPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 0*
CMPLTPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 1*

Table 5-12. Pseudo-Op and CMPPS Implementation

Pseudo-Op	CMPPS Implementation
CMPLEPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 2*
CMPUNORDPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 3*
CMPNEQPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 4*
CMPNLTPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 5*
CMPNLEPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 6*
CMPORDPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 7*

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

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 5-13, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPS instruction. See Table 5-13, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

Table 5-13. Pseudo-Op and VCMPPS Implementation

Pseudo-Op	CMPPS Implementation
VCMPEQPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 0*
VCMPLTPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 1*
VCMPLEPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 2*
VCMPUNORDPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 3*
VCMPNEQPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 4*
VCMPNLTPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 5*
VCMPNLEPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 6*
VCMPORDPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 7*
VCMPEQ_UQPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 8*
VCMPNGEPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 9*
VCMPNGTPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 0AH*

Table 5-13. Pseudo-Op and VCMPPS Implementation

Pseudo-Op	CMPPS Implementation
VCMPFALSEPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 0BH*
VCMPNEQ_OQPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 0CH*
VCMPGEPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 0DH*
VCMPGTPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 0EH*
VCMPTRUEPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 0FH*
VCMPEQ_OSPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 10H*
VCMPLT_OQPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 11H*
VCMPLE_OQPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 12H*
VCMPUNORD_SPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 13H*
VCMPNEQ_USPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 14H*
VCMPNLT_UQPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 15H*
VCMPNLE_UQPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 16H*
VCMPORD_SPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 17H*
VCMPEQ_USPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 18H*
VCMPNGE_UQPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 19H*
VCMPNGT_UQPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 1AH*
VCMPFALSE_OSPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 1BH*
VCMPNEQ_OSPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 1CH*
VCMPGE_OQPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 1DH*
VCMPGT_OQPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 1EH*
VCMPTRUE_USPS r*eg1, reg2, reg3*	VCMPPS r*eg1, reg2, reg3, 1FH*

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP3 ← EQ_OQ; OP5 ← EQ_OQ;
    1: OP3 ← LT_OS; OP5 ← LT_OS;
    2: OP3 ← LE_OS; OP5 ← LE_OS;
    3: OP3 ← UNORD_Q; OP5 ← UNORD_Q;
    4: OP3 ← NEQ_UQ; OP5 ← NEQ_UQ;
    5: OP3 ← NLT_US; OP5 ← NLT_US;
    6: OP3 ← NLE_US; OP5 ← NLE_US;
    7: OP3 ← ORD_Q; OP5 ← ORD_Q;
```

```
    8: OP5 ← EQ_UQ;
    9: OP5 ← NGE_US;
    10: OP5 ← NGT_US;
    11: OP5 ← FALSE_OQ;
    12: OP5 ← NEQ_OQ;
    13: OP5 ← GE_OS;
    14: OP5 ← GT_OS;
    15: OP5 ← TRUE_UQ;
    16: OP5 ← EQ_OS;
    17: OP5 ← LT_OQ;
    18: OP5 ← LE_OQ;
    19: OP5 ← UNORD_S;
    20: OP5 ← NEQ_US;
    21: OP5 ← NLT_UQ;
    22: OP5 ← NLE_UQ;
    23: OP5 ← ORD_S;
    24: OP5 ← EQ_US;
    25: OP5 ← NGE_UQ;
    26: OP5 ← NGT_UQ;
    27: OP5 ← FALSE_OS;
    28: OP5 ← NEQ_OS;
    29: OP5 ← GE_OQ;
    30: OP5 ← GT_OQ;
    31: OP5 ← TRUE_US;
    DEFAULT: Reserved
ESAC;
```

VCMPPS (VEX.256 encoded version)

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

```
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 000000000H; FI;
IF CMP4 = TRUE
    THEN DEST[159:128] ← FFFFFFFFH;
    ELSE DEST[159:128] ← 000000000H; FI;
IF CMP5 = TRUE
    THEN DEST[191:160] ← FFFFFFFFH;
    ELSE DEST[191:160] ← 000000000H; FI;
IF CMP6 = TRUE
    THEN DEST[223:192] ← FFFFFFFFH;
    ELSE DEST[223:192] ←000000000H; FI;
IF CMP7 = TRUE
    THEN DEST[255:224] ← FFFFFFFFH;
    ELSE DEST[255:224] ← 000000000H; FI;
```

VCMPPS (VEX.128 encoded version)

```
CMP0 ← SRC1[31:0] OP5 SRC2[31:0];
CMP1 ← SRC1[63:32] OP5 SRC2[63:32];
CMP2 ← SRC1[95:64] OP5 SRC2[95:64];
CMP3 ← SRC1[127:96] OP5 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ← 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ←000000000H; FI;
DEST[255:128] ← 0
```

CMPPS (128-bit Legacy SSE version)

```
CMP0 ← SRC1[31:0] OP3 SRC2[31:0];
CMP1 ← SRC1[63:32] OP3 SRC2[63:32];
CMP2 ← SRC1[95:64] OP3 SRC2[95:64];
CMP3 ← SRC1[127:96] OP3 SRC2[127:96];
IF CMP0 = TRUE
```

```
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ← 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ←000000000H; FI;
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCMPPS __m256 _mm256_cmp_ps(__m256 a, __m256 b, const int imm)

VCMPPS __m128 _mm_cmp_ps(__m128 a, __m128 b, const int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 5-9.

Denormal

Other Exceptions

See Exceptions Type 2

CMPSD- Compare Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F C2 /r ib CMPSD xmm1, xmm2/m64, imm8	A	V/V	SSE2	Compare low double precision floating-point value in xmm2/m64 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.NDS.LIG.F2.0F.WIG C2 /r ib VCMPSD xmm1, xmm2, xmm3/m64, imm8	B	V/V	AVX	Compare low double precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low double-precision floating-point values in the second source operand and the first source operand and returns the results in of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 64-bit memory location. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 64-bit memory location. The result is stored in the low quadword of the destination operand; the high quadword is filled with the contents of the high quadword of the first source operand. Bits (255:128) of the destination YMM register are zeroed.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 5-9). Bits 5 through 7 of the immediate are reserved.

- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 5-9). Bits 3 through 7 of the immediate are reserved.

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

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

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

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 5-14. Compiler should treat reserved Imm8 values as illegal syntax.

Table 5-14. Pseudo-Op and CMPSD Implementation

Pseudo-Op	CMPSD Implementation
CMPEQSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 0*
CMPLTSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 1*
CMPLESD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 2*
CMPUNORDSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 3*
CMPNEQSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 4*
CMPNLTSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 5*
CMPNLESD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 6*
CMPORDSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 7*

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

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 5-15, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 5-15, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

Table 5-15. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
VCMPEQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 0*
VCMPLTSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 1*
VCMPLESD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 2*
VCMPUNORDSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 3*
VCMPNEQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 4*
VCMPNLTSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 5*
VCMPNLESD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 6*
VCMPORDSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 7*
VCMPEQ_UQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 8*
VCMPNGESD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 9*
VCMPNGTSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 0AH*
VCMPFALSESD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 0BH*
VCMPNEQ_OQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 0CH*
VCMPGESD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 0DH*
VCMPGTSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 0EH*
VCMPTRUESD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 0FH*
VCMPEQ_OSSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 10H*
VCMPLT_OQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 11H*
VCMPLE_OQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 12H*
VCMPUNORD_SSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 13H*
VCMPNEQ_USSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 14H*
VCMPNLT_UQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 15H*
VCMPNLE_UQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 16H*
VCMPORD_SSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 17H*
VCMPEQ_USSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 18H*

Table 5-15. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
VCMPNGE_UQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 19H*
VCMPNGT_UQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 1AH*
VCMPFALSE_OSSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 1BH*
VCMPNEQ_OSSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 1CH*
VCMPGE_OQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 1DH*
VCMPGT_OQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 1EH*
VCMPTRUE_USSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 1FH*

Operation

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

```
    29: OP5 ← GE_OQ;
    30: OP5 ← GT_OQ;
    31: OP5 ← TRUE_US;
    DEFAULT: Reserved
ESAC;
```

CMPSD (128-bit Legacy SSE version)

```
CMP0 ← DEST[63:0] OP3 SRC[63:0];
IF CMP0 = TRUE
THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ← 0000000000000000H; FI;
DEST[255:64] (Unmodified)
```

VCMPSD (VEX.128 encoded version)

```
CMP0 ← SRC1[63:0] OP5 SRC2[63:0];
IF CMP0 = TRUE
THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ← 0000000000000000H; FI;
DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCMPSD __m128 _mm_cmp_sd(__m128 a, __m128 b, const int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 5-9 Denormal.

Other Exceptions

See Exceptions Type 3

CMPSS- Compare Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F C2 /r ib CMPSS xmm1, xmm2/m32, imm8	A	V/V	SSE	Compare low single precision floating-point value in xmm2/m32 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.NDS.LIG.F3.0F.WIG C2 /r ib VCMPSS xmm1, xmm2, xmm3/m32, imm8	B	V/V	AVX	Compare low single precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low single-precision floating-point values in the second source operand and the first source operand and returns the results of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 32-bit memory location. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 32-bit memory location. The result is stored in the low 32 bits of the destination operand; bits 128:32 of the destination operand are copied from the first source operand. Bits (255:128) of the destination YMM register are zeroed.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 5-9). Bits 5 through 7 of the immediate are reserved.

- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 5-9). Bits 3 through 7 of the immediate are reserved.

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

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

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

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 5-16. Compiler should treat reserved Imm8 values as illegal syntax.

Table 5-16. Pseudo-Op and CMPSS Implementation

Pseudo-Op	CMPSS Implementation
CMPEQSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 0*
CMPLTSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 1*
CMPLESS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 2*
CMPUNORDSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 3*
CMPNEQSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 4*
CMPNLTSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 5*
CMPNLESS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 6*
CMPORDSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 7*

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

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 5-15, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSS instruction. See Table 5-17, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

Table 5-17. Pseudo-Op and VCMPSS Implementation

Pseudo-Op	CMPSS Implementation
VCMPEQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 0*
VCMPLTSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 1*
VCMPLESS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 2*
VCMPUNORDSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 3*
VCMPNEQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 4*
VCMPNLTSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 5*
VCMPNLESS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 6*
VCMPORDSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 7*
VCMPEQ_UQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 8*
VCMPNGESS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 9*
VCMPNGTSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 0AH*
VCMPFALSESS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 0BH*
VCMPNEQ_OQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 0CH*
VCMPGESS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 0DH*
VCMPGTSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 0EH*
VCMPTRUESS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 0FH*
VCMPEQ_OSSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 10H*
VCMPLT_OQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 11H*
VCMPLE_OQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 12H*
VCMPUNORD_SSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 13H*
VCMPNEQ_USSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 14H*
VCMPNLT_UQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 15H*
VCMPNLE_UQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 16H*
VCMPORD_SSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 17H*
VCMPEQ_USSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 18H*

Table 5-17. Pseudo-Op and VCMPSS Implementation

Pseudo-Op	CMPSS Implementation
VCMPNGE_UQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 19H*
VCMPNGT_UQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 1AH*
VCMPFALSE_OSSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 1BH*
VCMPNEQ_OSSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 1CH*
VCMPGE_OQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 1DH*
VCMPGT_OQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 1EH*
VCMPTRUE_USSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 1FH*

Operation

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

```
    29: OP5 ← GE_OQ;
    30: OP5 ← GT_OQ;
    31: OP5 ← TRUE_US;
    DEFAULT: Reserved
ESAC;
```

CMPSS (128-bit Legacy SSE version)

```
CMP0 ← DEST[31:0] OP3 SRC[31:0];
IF CMP0 = TRUE
THEN DEST[31:0] ← FFFFFFFFH;
ELSE DEST[31:0] ← 00000000H; FI;
DEST[255:32] (Unmodified)
```

VCMPSS (VEX.128 encoded version)

```
CMP0 ← SRC1[31:0] OP5 SRC2[31:0];
IF CMP0 = TRUE
THEN DEST[31:0] ← FFFFFFFFH;
ELSE DEST[31:0] ← 00000000H; FI;
DEST[127:32] ← SRC1[127:32]
DEST[255:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCMPSS __m128 _mm_cmp_ss(__m128 a, __m128 b, const int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 5-9, Denormal.

Other Exceptions

See Exceptions Type 3

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

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2F /r COMISD xmm1, xmm2/m64	A	V/V	SSE2	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2F /r VCOMISD xmm1, xmm2/m64	A	V/V	AVX	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

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

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

COMISD (all versions)

```
RESULT ← OrderedCompare(DEST[63:0] <> SRC[63:0]) {
```

```
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF ← 111;
    GREATER_THAN: ZF,PF,CF ← 000;
    LESS_THAN: ZF,PF,CF ← 001;
    EQUAL: ZF,PF,CF ← 100;
ESAC;
OF, AF, SF ← 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_comieq_sd (__m128d a, __m128d b)

int _mm_comilt_sd (__m128d a, __m128d b)

int _mm_comile_sd (__m128d a, __m128d b)

int _mm_comigt_sd (__m128d a, __m128d b)

int _mm_comige_sd (__m128d a, __m128d b)

int _mm_comineq_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

See Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

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

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 2F /r COMISS xmm1, xmm2/m32	A	V/V	SSE	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.0F 2F.WIG /r **VCOMISS xmm1, xmm2/m32**	A	V/V	AVX	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

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

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

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

COMISS (all versions)

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

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_comieq_ss (__m128 a, __m128 b)

int _mm_comilt_ss (__m128 a, __m128 b)

int _mm_comile_ss (__m128 a, __m128 b)

int _mm_comigt_ss (__m128 a, __m128 b)

int _mm_comige_ss (__m128 a, __m128 b)

int _mm_comineq_ss (__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

See Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

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

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F E6 /r CVTDQ2PD xmm1, xmm2/m64	A	V/V	SSE2	Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1
VEX.128.F3.0F.WIG E6 /r VCVTDQ2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1
VEX.256.F3.0F.WIG E6 /r VCVTDQ2PD ymm1, xmm2/m128	A	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed double-precision floating-point values in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

VEX.128 encoded version: The source operand is an XMM register or 64- bit memory location. The destination operation is a YMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64- bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are unmodified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

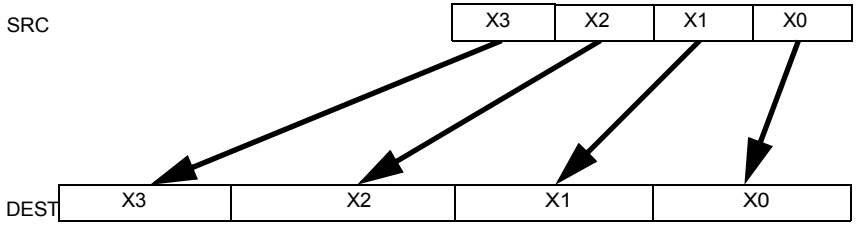


Figure 5-5. CVTDQ2PD (VEX.256 encoded version)

Operation

VCVTDQ2PD (VEX.256 encoded version)

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[127:96)

VCVTDQ2PD (VEX.128 encoded version)

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[255:128] ← 0

CVTDQ2PD (128-bit Legacy SSE version)

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[255:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTDQ2PD __m256d _mm256_cvtepi32_pd (__m128i src)

CVTDQ2PD __m128d _mm_cvtepi32_pd (__m128i src)

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.vvvv != 1111B.

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

Opcode Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 5B /r CVTDQ2PS xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1
VEX.128.0F.WIG 5B /r VCVTDQ2PS xmm1, xmm2/m128	A	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1
VEX.256.0F.WIG 5B /r VCVTDQ2PS ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed signed doubleword integers from ymm2/mem to eight packed single-precision floating-point values in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operation is a YMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are unmodified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTDQ2PS (VEX.256 encoded version)

DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96)
DEST[159:128] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[159:128])
DEST[191:160] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[191:160])
DEST[223:192] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[223:192])
DEST[255:224] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[255:224)

VCVTDQ2PS (VEX.128 encoded version)

DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96)
DEST[255:128] ← 0

CVTDQ2PS (128-bit Legacy SSE version)

DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96)
DEST[255:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTDQ2PS __m256 _mm256_cvtepi32_ps (__m256i src)

CVTDQ2PS __m128 _mm_cvtepi32_ps (__m128i src)

SIMD Floating-Point Exceptions

Precision

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

CVTPD2DQ- Convert Packed Double-Precision Floating-point values to Packed Doubleword Integers

Opcode Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F E6 /r CVTPD2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1
VEX.128.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1
VEX.256.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operation is a YMM register. The upper bits (255:64) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operation is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

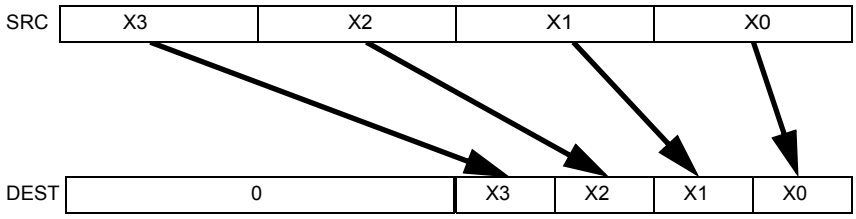


Figure 5-6. VCVTPD2DQ (VEX.256 encoded version)

Operation

VCVTPD2DQ (VEX.256 encoded version)

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[255:192)
DEST[255:128]← 0

VCVTPD2DQ (VEX.128 encoded version)

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[255:64]← 0

CVTPD2DQ (128-bit Legacy SSE version)

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[127:64] ← 0
DEST[255:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2DQ __m128i _mm256_cvtpd_epi32 (__m256d src)

CVTPD2DQ __m128i _mm_cvtpd_epi32 (__m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

CVTPD2PS- Convert Packed Double-Precision Floating-point values to Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5A /r CVTPD2PS xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double-precision floating-point values in xmm2/mem to two single-precision floating-point values in xmm1
VEX.128.66.0F.WIG 5A /r VCVTPD2PS xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double-precision floating-point values in xmm2/mem to two single-precision floating-point values in xmm1
VEX.256.66.0F.WIG 5A /r VCVTPD2PS xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double-precision floating-point values in ymm2/mem to four single-precision floating-point values in xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operation is a YMM register. The upper bits (255:64) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operation is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

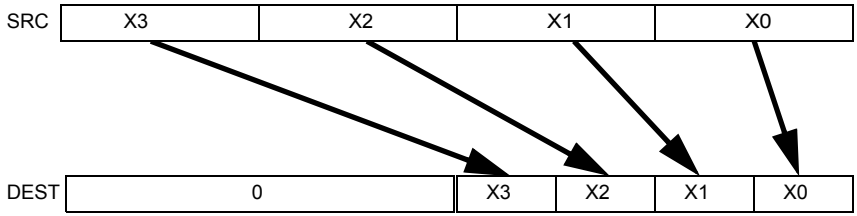


Figure 5-7. VCVTPD2PS (VEX.256 encoded version)

Operation

VCVTPD2PS (VEX.256 encoded version)

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[255:192)
DEST[255:128]← 0

VCVTPD2PS (VEX.128 encoded version)

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[255:64] ← 0

CVTPD2PS (128-bit Legacy SSE version)

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[127:64] ← 0
DEST[255:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2PS __m256 _mm256_cvtpd_ps (__m256d a)

CVTPD2PS __m128 _mm_cvtpd_ps (__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision, Underflow, Overflow, Denormal

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

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

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5B /r CVTPS2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1
VEX.128.66.0F.WIG 5B /r VCVTPS2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1
VEX.256.66.0F.WIG 5B /r VCVTPS2DQ ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed single precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operation is a YMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are unmodified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPS2DQ (VEX.256 encoded version)

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

VCVTPS2DQ (VEX.128 encoded version)

DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[255:128] ← 0

CVTPS2DQ (128-bit Legacy SSE version)

DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[255:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2DQ __ m256i _mm256_cvtps_epi32 (__m256 a)

CVTPS2DQ __m128i _mm_cvtps_epi32 (__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

CVTPS2PD- Convert Packed Single Precision Floating-point values to Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 5A /r CVTPS2PD xmm1, xmm2/m64	A	V/V	SSE2	Convert two packed single-precision floating-point values in xmm2/mem to two packed double-precision floating-point values in xmm1
VEX.128.0F.WIG 5A /r VCVTPS2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed single-precision floating-point values in xmm2/mem to two packed double-precision floating-point values in xmm1
VEX.256.0F.WIG 5A /r VCVTPS2PD ymm1, xmm2/m128	A	V/V	AVX	Convert four packed single-precision floating-point values in xmm2/mem to four packed double-precision floating-point values in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

VEX.128 encoded version: The source operand is an XMM register or 64- bit memory location. The destination operation is a YMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64- bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are unmodified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

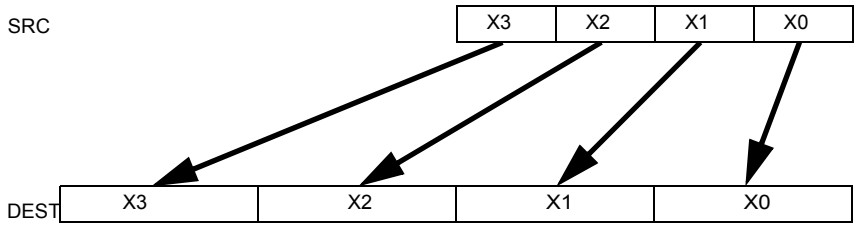


Figure 5-8. CVTPS2PD (VEX.256 encoded version)

Operation

VCVTPS2PD (VEX.256 encoded version)

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[127:96)

VCVTPS2PD (VEX.128 encoded version)

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[255:128] ← 0

CVTPS2PD (128-bit Legacy SSE version)

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[255:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2PD __m256d _mm256_cvtps_pd (__m128 a)

CVTPS2PD __m128d _mm_cvtps_pd (__m128 a)

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

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

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

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the

maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

(V)CVTSD2SI

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

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_cvtsd_si32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

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

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5A /r CVTSD2SS xmm1, xmm2/m64	A	V/V	SSE2	Convert one double-precision floating-point value in xmm2/m64 to one single-precision floating-point value in xmm1.
VEX.NDS.LIG.F2.0F.WIG 5A /r VCVTSD2SS xmm1,xmm2, xmm3/m64	B	V/V	AVX	Convert one double-precision floating-point value in xmm3/m64 to one single-precision floating-point value and merge with high bits in xmm2.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts a double-precision floating-point value in the second source operand to a single-precision floating-point value in the destination operand.

When the second source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand, and the upper 3 doublewords are copied from the upper 3 doublewords of the first source operand. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

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

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

Operation

VCVTSD2SS (VEX.128 encoded version)

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
DEST[127:32] ← SRC1[127:32]
DEST[255:128] ← 0

CVTSD2SS (128-bit Legacy SSE version)

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[255:32] Unmodified *)

Intel C/C++ Compiler Intrinsic Equivalent

CVTSD2SS __m128_mm_cvtsd_ss(__m128 a, __m128d b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

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

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

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the second source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

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

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

Operation

VCVTSI2SD

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0
```

CVTSI2SD

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

DEST[255:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTSI2SD __m128d _mm_cvtsi32_sd(__m128d a, int b)

SIMD Floating-Point Exceptions

Precision

Other Exceptions

See Exceptions Type 3

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

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

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

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

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

Operation

VCVTSI2SS (VEX.128 encoded version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] ← SRC1[127:32]
DEST[255:128] ← 0
```

CVTSI2SS (128-bit Legacy SSE version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[255:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTSI2SS __m128_mm_cvtsi32_ss(__m128 a, int b)

SIMD Floating-Point Exceptions

Precision

Other Exceptions

See Exceptions Type 3

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

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5A /r CVTSS2SD xmm1, xmm2/m32	A	V/V	SSE2	Convert one single-precision floating-point value in xmm2/m32 to one double-precision floating-point value in xmm1.
VEX.NDS.LIG.F3.0F.WIG 5A /r VCVTSS2SD xmm1, xmm2, xmm3/m32	B	V/V	AVX	Convert one single-precision floating-point value in xmm3/m32 to one double-precision floating-point value and merge with high bits of xmm2.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts a single-precision floating-point value in the second source operand to a double-precision floating-point value in the destination operand. When the second source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand, and the high quadword is copied from the first source operand.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

Operation

VCVTSS2SD (VEX.128 encoded version)

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0])

DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0

CVTSS2SD (128-bit Legacy SSE version)
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[255:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

CVTSS2SD __m128d _mm_cvtss_sd(__m128d a, __m128 b)

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type 3

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

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

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

CVTSS2SI

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

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_cvtss_si32(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

CVTTPD2DQ- Convert with Truncation Packed Double-Precision Floating-point values to Packed Doubleword Integers

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E6 /r CVTTPD2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation
VEX.128.66.0F.WIG E6 /r VCVTTPD2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation
VEX.256.66.0F.WIG E6 /r VCVTTPD2DQ xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1 using truncation

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operation is a YMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are unmodified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

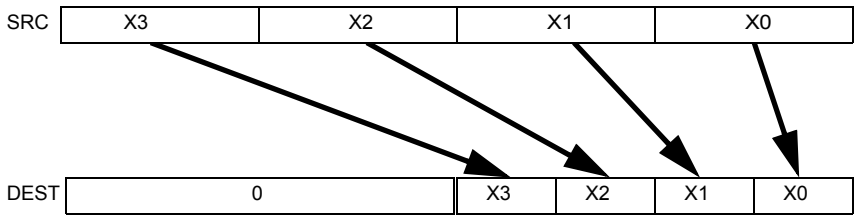


Figure 5-9. VCVTTPD2DQ (VEX.256 encoded version)

Operation

VCVTTPD2DQ (VEX.256 encoded version)

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[255:192)
DEST[255:128]← 0

VCVTTPD2DQ (VEX.128 encoded version)

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[255:64]← 0

CVTTPD2DQ (128-bit Legacy SSE version)

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[127:64] ← 0

DEST[255:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2DQ __m128i _mm256_cvttpd_epi32 (__m256d src)

CVTTDQ2PD __m128i _mm_cvttpd_epi32 (__m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

CVTTPS2DQ- Convert with Truncation Packed Single Precision Floating-Point Values to Packed Singed Doubleword Integer Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5B /r CVTTPS2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation
VEX.128.F3.0F.WIG 5B /r VCVTTPS2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation
VEX.256.F3.0F.WIG 5B /r VCVTTPS2DQ ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed single precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1 using truncation

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

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

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operation is a YMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are unmodified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTTPS2DQ (VEX.256 encoded version)

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

VCVTTPS2DQ (VEX.128 encoded version)

DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[255:128] ← 0

CVTTPS2DQ (128-bit Legacy SSE version)

DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[255:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2DQ __m256i _mm256_cvttps_epi32 (__m256 a)

CVTTPS2DQ __m128i _mm_cvttps_epi32 (__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

CVTTSD2SI- Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Doubleword Integer

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

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

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

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

CVTTSD2SI

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

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_cvttsd_si32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

CVTTSS2SI- Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer

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

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a single-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

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

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

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

CVTTSS2SI

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

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_cvttss_si32(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

DIVPD- Divide Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5E /r DIVPD xmm1, xmm2/m128	A	V/V	SSE2	Divide packed double-precision floating-point values in xmm1 by packed double-precision floating-point values in xmm2/mem
VEX.NDS.128.66.0F.WIG 5E /r VDIVPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Divide packed double-precision floating-point values in xmm2 by packed double-precision floating-point values in xmm3/mem
VEX.NDS.256.66.0F.WIG 5E /r VDIVPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Divide packed double-precision floating-point values in ymm2 by packed double-precision floating-point values in ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD divide of the two or four packed double-precision floating-point values in the first source operand by the two or four packed double-precision floating-point values in the second source operand.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VDIVPD (VEX.256 encoded version)
DEST[63:0] ← SRC1[63:0] / SRC2[63:0]
DEST[127:64] ← SRC1[127:64] / SRC2[127:64]
DEST[191:128] ← SRC1[191:128] / SRC2[191:128]
DEST[255:192] ← SRC1[255:192] / SRC2[255:192]

VDIVPD (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0] / SRC2[63:0]
DEST[127:64] ← SRC1[127:64] / SRC2[127:64]
DEST[255:128] ← 0
DIVPD (128-bit Legacy SSE version)
DEST[63:0] ← SRC1[63:0] / SRC2[63:0]
DEST[127:64] ← SRC1[127:64] / SRC2[127:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VDIVPD __m256d _mm256_div_pd (__m256d a, __m256d b);

DIVPD __m128d _mm_div_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

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

Other Exceptions

See Exceptions Type 2

DIVPS- Divide Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 5E /r DIVPS xmm1, xmm2/m128	A	V/V	SSE	Divide packed single-precision floating-point values in xmm1 by packed double-precision floating-point values in xmm2/mem
VEX.NDS.128.0F.WIG 5E /r VDIVPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Divide packed single-precision floating-point values in xmm2 by packed double-precision floating-point values in xmm3/mem
VEX.NDS.256.0F.WIG 5E /r VDIVPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Divide packed single-precision floating-point values in ymm2 by packed double-precision floating-point values in ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD divide of the four or eight packed single-precision floating-point values in the first source operand by the four or eight packed single-precision floating-point values in the second source operand.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VDIVPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
DEST[63:32] ← SRC1[63:32] / SRC2[63:32]
DEST[95:64] ← SRC1[95:64] / SRC2[95:64]
DEST[127:96] ← SRC1[127:96] / SRC2[127:96]
DEST[159:128] ← SRC1[159:128] / SRC2[159:128]
DEST[191:160]← SRC1[191:160] / SRC2[191:160]
DEST[223:192] ← SRC1[223:192] / SRC2[223:192]
DEST[255:224] ← SRC1[255:224] / SRC2[255:224].

VDIVPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
DEST[63:32] ← SRC1[63:32] / SRC2[63:32]
DEST[95:64] ← SRC1[95:64] / SRC2[95:64]
DEST[127:96] ← SRC1[127:96] / SRC2[127:96]
DEST[255:128] ← 0

DIVPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
DEST[63:32] ← SRC1[63:32] / SRC2[63:32]
DEST[95:64] ← SRC1[95:64] / SRC2[95:64]
DEST[127:96] ← SRC1[127:96] / SRC2[127:96]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VDIVPS __m256 _mm256_div_ps (__m256 a, __m256 b);

DIVPS __m128 _mm_div_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

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

Other Exceptions

See Exceptions Type 2

DIVSD- Divide Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5E /r DIVSD xmm1, xmm2/m64	A	V/V	SSE2	Divide low double-precision floating point values in xmm1 by low double precision floating-point value in xmm2/mem64.
VEX.NDS.LIG.F2.0F.WIG 5E /r VDIVSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Divide low double-precision floating point values in xmm2 by low double precision floating-point value in xmm3/mem64.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Divides the low double-precision floating-point value in the first source operand by the low double-precision floating-point value in the second source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination hyperons are XMM registers. The high quadword of the destination operand is copied from the high quadword of the first source operand. See Chapter 11 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an overview of a scalar double-precision floating-point operation.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VDIVSD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0

DIVSD (128-bit Legacy SSE version)
DEST[63:0] ← DEST[63:0] / SRC[63:0]
DEST[255:64] (Unmodified)

Intel C/C++ 6Compiler Intrinsic Equivalent

DIVSD __m128d _mm_div_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

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

Other Exceptions

See Exceptions Type 3

DIVSS- Divide Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5E /r DIVSS xmm1, xmm2/m32	A	V/V	SSE	Divide low single-precision floating point value in xmm1 by low single precision floating-point value in xmm2/m32.
VEX.NDS.LIG.F3.0F.WIG 5E /r VDIVSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Divide low single-precision floating point value in xmm2 by low single precision floating-point value in xmm3/m32.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Divides the low single-precision floating-point value in the first source operand by the low single-precision floating-point value in the second source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. The three high-order doublewords of the destination are copied from the same dwords of the first source operand. See Chapter 10 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an overview of a scalar single-precision floating-point operation.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VDIVSS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[255:128] ← 0

DIVSS (128-bit Legacy SSE version)
DEST[31:0] ← DEST[31:0] / SRC[31:0]
DEST[255:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

DIVSS __m128 _mm_div_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

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

Other Exceptions

See Exceptions Type 3

DPPD- Dot Product of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 41 /r ib DPPD xmm1, xmm3/m128, imm8	A	V/V	SSE4_1	Selectively multiply packed DP floating-point values from xmm1 with packed DP floating-point values from xmm2, add and selectively store the packed DP floating-point values to xmm1
VEX.NDS.128.66.0F3A.WIG 41 /r ib VDPPD xmm1,xmm2, xmm3/m128, imm8	B	V/V	AVX	Selectively multiply packed DP floating-point values from xmm2 with packed DP floating-point values from xmm3, add and selectively store the packed DP floating-point values to xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Conditionally multiplies the packed double precision floating-point values in the destination operand (first operand) with the packed double-precision floating-point values in the source (second operand) depending on a mask extracted from bits 4-5 of the immediate operand. Each of the two resulting double-precision values is summed and this sum is conditionally broadcast to each of 2 positions in the destination operand if the corresponding bit of the mask selected from bits 0-1 of the immediate operand is "1". If the corresponding low bit 0-1 of the mask is zero, the destination is set to zero. DPPD follows the NaN forwarding rules stated in the Software Developer's Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationaly generated NaNs will have at least one NaN propagated to the destination.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

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

Operation

DP_primitive (SRC1, SRC2)

```
IF (imm8[4] == 1) THEN Temp1[63:0] ← SRC1[63:0] * SRC2[63:0];
ELSE Temp1[63:0] ← +0.0;
IF (imm8[5] == 1) THEN Temp1[127:64] ← SRC1[127:64] * SRC2[127:64];
ELSE Temp1[127:64] ← +0.0;
Temp2[63:0] ← Temp1[63:0] + Temp1[127:64];
IF (imm8[0] == 1) THEN DEST[63:0] ← Temp2[63:0];
ELSE DEST[63:0] ← +0.0;
IF (imm8[1] == 1) THEN DEST[127:64] ← Temp2[63:0];
ELSE DEST[127:64] ← +0.0;
```

VDPPD (VEX.128 encoded version)

```
DEST[127:0]←DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[255:128] ← 0
```

DPPD (128-bit Legacy SSE version)

```
DEST[127:0]←DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

DPPD __m128d _mm_dp_pd (__m128d a, __m128d b, const int mask);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Exceptions are determined separately for each add and multiply operation. Unmasked exceptions will leave the destination untouched

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.L=1

DPPS- Dot Product of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 40 /r ib DPPS xmm1, xmm3/m128, imm8	A	V/V	SSE4_1	Multiply packed SP floating point values from xmm1 with packed SP floating point values from xmm3/mem selectively add and store to xmm1
VEX.NDS.128.66.0F3A.WIG 40 /r ib VDPPS xmm1,xmm2, xmm3/m128, imm8	B	V/V	AVX	Multiply packed SP floating point values from xmm1 with packed SP floating point values from xmm2/mem selectively add and store to xmm1
VEX.NDS.256.66.0F3A.WIG 40 /r ib VDPPS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Multiply packed single-precision floating-point values from ymm2 with packed SP floating point values from ymm3/mem, selectively add pairs of elements and store to ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the packed single precision floating point values in the first source operand (second operand) with the packed single-precision floats in the second source (third operand). Each of the four resulting single-precision values is conditionally summed depending on a mask extracted from the high 4 bits of the immediate operand. This sum is broadcast to each of 4 positions in the destination operand (first operand) if the corresponding bit of the mask selected from the low 4 bits of the immediate operand is "1". If the corresponding low bit 0-3 of the mask is zero, the destination is set to zero.

The process is replicated for the high elements of the destination YMM.

DPPS follows the NaN forwarding rules stated in the Software Developer's Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal

propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationaly generated NaNs will have at least one NaN propagated to the destination.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

DP_primitive (SRC1, SRC2)

```
IF (imm8[4] == 1) THEN Temp1[31:0] ← SRC1[31:0] * SRC2[31:0];
    ELSE Temp1[31:0] ← +0.0;
IF (imm8[5] == 1) THEN Temp1[63:32] ← SRC1[63:32] * SRC[63:32];
    ELSE Temp1[63:32] ← +0.0;
IF (imm8[6] == 1) THEN Temp1[95:64] ← SRC1[95:64] * SRC2[95:64];
    ELSE Temp1[95:64] ← +0.0;
IF (imm8[7] == 1) THEN Temp1[127:96] ← SRC1[127:96] * SRC2[127:96];
    ELSE Temp1[127:96] ← +0.0;

Temp2[31:0] ← Temp1[31:0] + Temp1[63:32];
Temp3[31:0] ← Temp1[95:64] + Temp1[127:96];
Temp4[31:0] ← Temp2[31:0] + Temp3[31:0];

IF (imm8[0] == 1) THEN DEST[31:0] ← Temp4[31:0];
    ELSE DEST[31:0] ← +0.0;
IF (imm8[1] == 1) THEN DEST[63:32] ← Temp4[31:0];
    ELSE DEST[63:32] ← +0.0;
IF (imm8[2] == 1) THEN DEST[95:64] ← Temp4[31:0];
    ELSE DEST[95:64] ← +0.0;
IF (imm8[3] == 1) THEN DEST[127:96] ← Temp4[31:0];
    ELSE DEST[127:96] ← +0.0;
```

VDPPS (VEX.256 encoded version)

```
DEST[127:0]←DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[255:128]←DP_Primitive(SRC1[255:128], SRC2[255:128]);
```

VDPPS (VEX.128 encoded version)
DEST[127:0]←DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[255:128] ← 0

DPP (128-bit Legacy SSE version)
DEST[127:0]←DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VDPPS __m256 _mm256_dp_ps (__m256 a, __m256 b, const int mask);

(V)DPPS __m128 _mm_dp_ps (__m128 a, __m128 b, const int mask);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Exceptions are determined separately for each add and multiply operation, in the order of their execution. Unmasked exceptions will leave the destination operands unchanged.

Other Exceptions

See Exceptions Type 2

VEXTRACTF128- Extract packed floating-point values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 19 /r ib VEXTRACTF128 xmm1/m128, ymm2, imm8	A	V/V	AVX	Extract 128 bits of packed floating-point values from ymm2 and store results in xmm1/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Extracts 128-bits of packed floating-point values from the source operand (second operand) at an 128-bit offset from imm8[0] into the destination operand (first operand). The destination may be either an XMM register or an 128-bit memory location.

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

The high 7 bits of the immediate are ignored.

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

Operation

VEXTRACTF128 (memory destination form)

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

VEXTRACTF128 (register destination form)

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

Intel C/C++ Compiler Intrinsic Equivalent

VEXTRACTF128 __m128 _mm256_extractf128_ps (__m256 a, int offset);

VEXTRACTF128 __m128d _mm256_extractf128_pd (__m256d a, int offset);

VEXTRACTF128 __m128i_mm256_extractf128_si256(__m256i a, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	IF VEX.L = 0
	If VEX.W = 1

EXTRACTPS- Extract packed floating-point values

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

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

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

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

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

VEX.128 encoded version: When VEX.128.66.0F3A.W1 17 form is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits. VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

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

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

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

Operation

VEXTRACTPS (VEX.128 encoded version)

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

EXTRACTPS (128-bit Legacy SSE version)

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

Intel C/C++ Compiler Intrinsic Equivalent

EXTRACTPS _mm_extractmem_ps (float *dest, __m128 a, const int nidx);

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; Additionally

#UD	IF VEX.L = 1

HADDPD- Add Horizontal Double Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 7C /r HADDPD xmm1, xmm2/m128	A	V/V	SSE3	Horizontal add packed double-precision floating-point values from xmm1 and xmm2/mem
VEX.NDS.128.66.0F.WIG 7C /r VHADDPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Horizontal add packed double-precision floating-point values from xmm2 and xmm3/mem
VEX.NDS.256.66.0F.WIG 7C /r VHADDPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Horizontal add packed double-precision floating-point values from ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Adds pairs of adjacent double-precision floating-point values in the first source operand and second source operand and stores results in the destination.

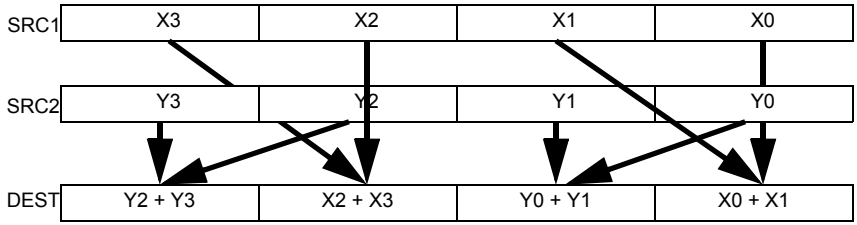


Figure 5-10. VHADDPD operation

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VHADDPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[191:128] ← SRC1[255:192] + SRC1[191:128]
DEST[255:192] ← SRC2[255:192] + SRC2[191:128]

VHADDPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[255:128] ← 0

HADDPD (128-bit Legacy SSE version)

DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VHADDPD __m256d _mm256_hadd_pd (__m256d a, __m256d b);

HADDPD __m128d _mm_hadd_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

HADDPS- Add Horizontal Single Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 7C /r HADDPS xmm1, xmm2/m128	A	V/V	SSE3	Horizontal add packed single-precision floating-point values from xmm1 and xmm2/mem
VEX.NDS.128.F2.0F.WIG 7C /r VHADDPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Horizontal add packed single-precision floating-point values from xmm2 and xmm3/mem
VEX.NDS.256.F2.0F.WIG 7C /r VHADDPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Horizontal add packed single-precision floating-point values from ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Adds pairs of adjacent single-precision floating-point values in the first source operand and second source operand and stores results in the destination.

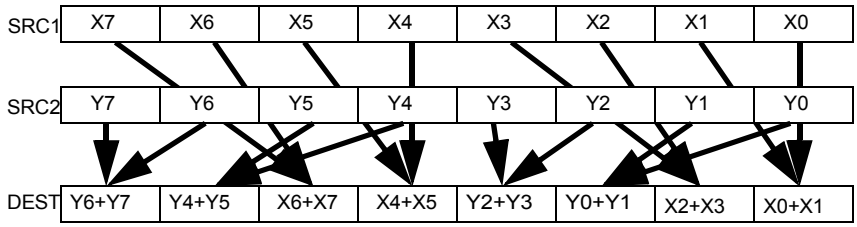


Figure 5-11. VHADDPS operation

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VHADDPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[63:32] + SRC1[31:0]
DEST[63:32] ← SRC1[127:96] + SRC1[95:64]
DEST[95:64] ← SRC2[63:32] + SRC2[31:0]
DEST[127:96] ← SRC2[127:96] + SRC2[95:64]
DEST[159:128] ← SRC1[191:160] + SRC1[159:128]
DEST[191:160] ← SRC1[255:224] + SRC1[223:192]
DEST[223:192] ← SRC2[191:160] + SRC2[159:128]
DEST[255:224] ← SRC2[255:224] + SRC2[223:192]

VHADDPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[63:32] + SRC1[31:0]
DEST[63:32] ← SRC1[127:96] + SRC1[95:64]
DEST[95:64] ← SRC2[63:32] + SRC2[31:0]
DEST[127:96] ← SRC2[127:96] + SRC2[95:64]

DEST[255:128] ← 0

HADDPS (128-bit Legacy SSE version)
DEST[31:0] ← SRC1[63:32] + SRC1[31:0]
DEST[63:32] ← SRC1[127:96] + SRC1[95:64]
DEST[95:64] ← SRC2[63:32] + SRC2[31:0]
DEST[127:96] ← SRC2[127:96] + SRC2[95:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VHADDPS __m256 _mm256_hadd_ps (__m256 a, __m256 b);

HADDPS __m128 _mm_hadd_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

HSUBPD- Subtract Horizontal Double Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 7D /r HSUBPD xmm1, xmm2/m128	A	V/V	SSE3	Horizontal subtract packed double-precision floating-point values from xmm1 and xmm2/mem
VEX.NDS.128.66.0F.WIG 7D /r VHSUBPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Horizontal subtract packed double-precision floating-point values from xmm2 and xmm3/mem
VEX.NDS.256.66.0F.WIG 7D /r VHSUBPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Horizontal subtract packed double-precision floating-point values from ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Subtract pairs of adjacent double-precision floating-point values in the first source operand and second source operand and stores results in the destination.

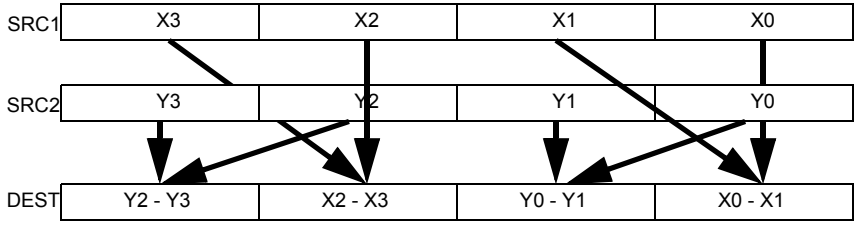


Figure 5-12. VHSUBPD operation

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VHSUBPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[63:0] - SRC1[127:64]
DEST[127:64] ← SRC2[63:0] - SRC2[127:64]
DEST[191:128] ← SRC1[191:128] - SRC1[255:192]
DEST[255:192] ← SRC2[191:128] - SRC2[255:192]

VHSUBPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] - SRC1[127:64]
DEST[127:64] ← SRC2[63:0] - SRC2[127:64]
DEST[255:128] ← 0

HSUBPD (128-bit Legacy SSE version)

DEST[63:0] ← SRC1[63:0] - SRC1[127:64]
DEST[127:64] ← SRC2[63:0] - SRC2[127:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VHSUBPD __m256d _mm256_hsub_pd (__m256d a, __m256d b);

HSUBPD __m128d _mm_hsub_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

HSUBPS- Subtract Horizontal Single Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 7D /r HSUBPS xmm1, xmm2/m128	A	V/V	SSE3	Horizontal subtract packed single-precision floating-point values from xmm1 and xmm2/mem
VEX.NDS.128.F2.0F.WIG 7D /r VHSUBPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Horizontal subtract packed single-precision floating-point values from xmm2 and xmm3/mem
VEX.NDS.256.F2.0F.WIG 7D /r VHSUBPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Horizontal subtract packed single-precision floating-point values from ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Subtract pairs of adjacent single-precision floating-point values in the first source operand and second source operand and stores results in the destination.

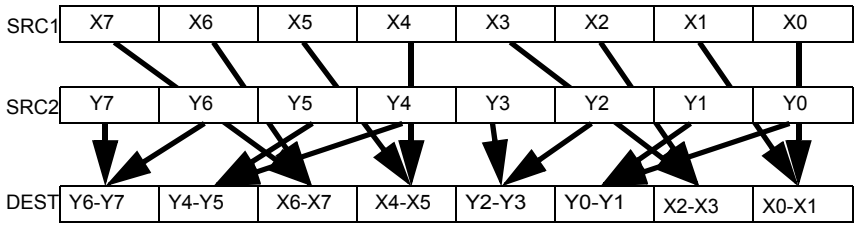


Figure 5-13. VHSUBPS operation

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VHSUBPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[31:0] - SRC1[63:32]
DEST[63:32] ← SRC1[95:64] - SRC1[127:96]
DEST[95:64] ← SRC2[31:0] - SRC2[63:32]
DEST[127:96] ← SRC2[95:64] - SRC2[127:96]
DEST[159:128] ← SRC1[159:128] - SRC1[191:160]
DEST[191:160] ← SRC1[223:192] - SRC1[255:224]
DEST[223:192] ← SRC2[159:128] - SRC2[191:160]
DEST[255:224] ← SRC2[223:192] - SRC2[255:224]

VHSUBPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] - SRC1[63:32]
DEST[63:32] ← SRC1[95:64] - SRC1[127:96]
DEST[95:64] ← SRC2[31:0] - SRC2[63:32]
DEST[127:96] ← SRC2[95:64] - SRC2[127:96]

DEST[255:128] ← 0

HSUBPS (128-bit Legacy SSE version)
DEST[31:0] ← SRC1[31:0] - SRC1[63:32]
DEST[63:32] ← SRC1[95:64] - SRC1[127:96]
DEST[95:64] ← SRC2[31:0] - SRC2[63:32]
DEST[127:96] ← SRC2[95:64] - SRC2[127:96]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VHSUBPS __m256 _mm256_hsub_ps (__m256 a, __m256 b);

HSUBPS __m128 _mm_hsub_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VINSERTF128- Insert packed floating-point values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 18 /r ib VINSERTF128 ymm1, ymm2, xmm3/m128, imm8	A	V/V	AVX	Insert 128-bits of packed floating-point values from xmm3/mem and the remaining values from ymm2 into ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an insertion of 128-bits of packed floating-point values from the second source operand (third operand) into an the destination operand (first operand) at an 128-bit offset from imm8[0]. The remaining portions of the destination are written by the corresponding fields of the first source operand (second operand). The second source operand can be either an XMM register or a 128-bit memory location.

The high 7 bits of the immediate are ignored.

Operation

INSERTF128

```
TEMP[255:0] ← SRC1[255:0]
CASE (imm8[0]) OF
    0: TEMP[127:0] ← SRC2[127:0]
    1: TEMP[255:128] ← SRC2[127:0]
ESAC
DEST ←TEMP
```

Intel C/C++ Compiler Intrinsic Equivalent

INSERTF128 __m256 _mm256_insertf128_ps (__m256 a, __m128 b, int offset);

INSERTF128 __m256d _mm256_insertf128_pd (__m256d a, __m128d b, int offset);

INSERTF128 __m256i _mm256_insertf128_si256 (__m256i a, __m128i b, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.W = 1

INSERTPS- Insert Scalar Single Precision Floating-Point Value

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

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

(register source form)

Select a single precision floating-point element from second source as indicated by Count_S bits of the immediate operand and insert it into the first source at the location indicated by the Count_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

(memory source form)

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

128-bit Legacy SSE version: The first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version. The destination and first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

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

Operation

VINSERTPS (VEX.128 encoded version)

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

IF (ZMASK[0] == 1) THEN DEST[31:0] ← 00000000H
    ELSE DEST[31:0] ← TMP2[31:0]
IF (ZMASK[1] == 1) THEN DEST[63:32] ← 00000000H
    ELSE DEST[63:32] ← TMP2[63:32]
IF (ZMASK[2] == 1) THEN DEST[95:64] ← 00000000H
    ELSE DEST[95:64] ← TMP2[95:64]
```

```
IF (ZMASK[3] == 1) THEN DEST[127:96] ← 00000000H
    ELSE DEST[127:96] ← TMP2[127:96]
DEST[255:128] ← 0
```

INSERTPS (128-bit Legacy SSE version)

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

CASE (COUNT_D) OF
    0: TMP2[31:0] ← TMP
        TMP2[127:32] ← DEST[127:32]
    1: TMP2[63:32] ← TMP
        TMP2[31:0] ← DEST[31:0]
        TMP2[127:64] ← DEST[127:64]
    2: TMP2[95:64] ← TMP
        TMP2[63:0] ← DEST[63:0]
        TMP2[127:96] ← DEST[127:96]
    3: TMP2[127:96] ← TMP
        TMP2[95:0] ← DEST[95:0]
ESAC;

IF (ZMASK[0] == 1) THEN DEST[31:0] ← 00000000H
    ELSE DEST[31:0] ← TMP2[31:0]
IF (ZMASK[1] == 1) THEN DEST[63:32] ← 00000000H
    ELSE DEST[63:32] ← TMP2[63:32]
IF (ZMASK[2] == 1) THEN DEST[95:64] ← 00000000H
    ELSE DEST[95:64] ← TMP2[95:64]
IF (ZMASK[3] == 1) THEN DEST[127:96] ← 00000000H
    ELSE DEST[127:96] ← TMP2[127:96]
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5

LDDQU- Move Unaligned Integer

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F F0 /r LDDQU xmm1, m128	A	V/V	SSE3	Load unaligned packed integer values from mem to xmm1
VEX.128.F2.0F.WIG F0 /r VLDDQU xmm1, m128	A	V/V	AVX	Load unaligned packed integer values from mem to xmm1
VEX.256.F2.0F.WIG F0 /r VLDDQU ymm1, m256	A	V/V	AVX	Load unaligned packed integer values from mem to ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The instruction is functionally similar to VMOVDQU YMM, m256 for loading from memory. That is: 32 bytes of data starting at an address specified by the source memory operand (second operand) are fetched from memory and placed in a destination register (first operand). The source operand need not be aligned on a 32-byte boundary. Up to 64 bytes may be loaded from memory; this is implementation dependent.

This instruction may improve performance relative to VMOVDQU if the source operand crosses a cache line boundary. In situations that require the data loaded by VLDDQU be modified and stored to the same location, use VMOVDQU or VMOVDQA instead of VLDDQU. To move double quadwords to or from memory locations that are known to be aligned on 32-byte boundaries, use the VMOVDQA instruction.

Implementation Notes

• If the source is aligned to a 32-byte boundary, based on the implementation, the 32 bytes may be loaded more than once. For that reason, the usage of VLDDQU should be avoided when using uncached or write-combining (WC) memory regions. For uncached or WC memory regions, keep using VMOVDQU.

• This instruction is a replacement for VMOVDQU (load) in situations where cache line splits significantly affect performance. It should not be used in situations where store-load forwarding is performance critical. If performance of store-load forwarding is critical to the application, use VMOVDQA store-load pairs when data is 256-bit aligned or VMOVDQU store-load pairs when data is 256-bit unaligned.

• If the memory address is not aligned on 32-byte boundary, some implementations may load up to 64 bytes and return 32 bytes in the destination. Some processor implementations may issue multiple loads to access the appropriate 32 bytes. Developers of multi-threaded or multi-processor software should be aware that on these processors the loads will be performed in a non-atomic way.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VLDDQU (VEX.256 encoded version)
DEST[255:0] ← SRC[255:0]

VLDDQU (VEX.128 encoded version)
DEST[127:0] ← SRC[127:0]
DEST[255:128] ← 0

LDDQU (128-bit Legacy SSE version)
DEST[127:0] ← SRC[127:0]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

LDDQU __m128i _mm_lddqu_si128 (__m128i * p);

LDDQU __m256i _mm256_lddqu_si256 (__m256i * p);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4
Note treatment of #AC varies

VLDMXCSR—Load MXCSR Register

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.LZ.0F.WIG AE /2 VLDMXCSR *m32*	A	V/V	AVX	Load MXCSR register from *m32*.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (r)	NA	NA	NA

Description

Loads the source operand into the MXCSR control/status register. The source operand is a 32-bit memory location.

The VLDMXCSR instruction is typically used in conjunction with the VSTMXCSR instruction for software that use instruction set extensions operating on the YMM state.

The default MXCSR value at reset is 1F80H.

If a VLDMXCSR instruction clears a SIMD floating-point exception mask bit and sets the corresponding exception flag bit, a SIMD floating-point exception will not be immediately generated. The exception will be generated only upon the execution of the next instruction that meets both conditions below:

- the instruction must operate on an XMM or YMM register operand,
- the instruction causes that particular SIMD floating-point exception to be reported.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

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

Operation

```
MXCSR ← m32;
```

C/C++ Compiler Intrinsic Equivalent

_mm_setcsr(unsigned int i)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 9; additionally

#GP For an attempt to set reserved bits in MXCSR

MASKMOVDQU- Store Selected Bytes of Double Quadword with NT Hint

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F F7 /r MASKMOVDQU xmm1, xmm2	A	V/V	SSE2	Selectively write bytes from xmm1 to memory location using the byte mask in xmm2. The default memory location is specified by DS:DI/EDI/RDI
VEX.128.66.0F.WIG F7 /r VMASKMOVDQU xmm1, xmm2	A	V/V	AVX	Selectively write bytes from xmm1 to memory location using the byte mask in xmm2. The default memory location is specified by DS:DI/EDI/RDI

Instruction Operand Encoding[1]

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

Stores selected bytes from the source operand (first operand) into an 128-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are XMM registers. The location of the first byte of the memory location is specified by DI/EDI/RDI and DS registers. The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVDQU instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10, of the IA-32 Intel® Architecture Software Developer's Manual, Volume 1). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction

1. ModRM.MOD = 011B required

should be used in conjunction with MASKMOVDQU instructions if multiple processors might use different memory types to read/write the destination memory locations.

Behavior with a mask of all 0s is as follows:

• No data will be written to memory.

• Signaling of breakpoints (code or data) is not guaranteed; different processor implementations may signal or not signal these breakpoints.

• Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).

• If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation- specific.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

The MASKMOVDQU instruction can be used to improve performance of algorithms that need to merge data on a byte-by-byte basis. MASKMOVDQU should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the bytemask without allocating old data prior to the store.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

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

Operation

MASKMOVDQU

```
IF (MASK[7] = 1)
    THEN DEST[DS:DI/EDI/RDI] ← SRC[7:0] ELSE (* Memory location unchanged *); FI;
IF (MASK[15] = 1)
    THEN DEST[DS:DI/EDI/RDI +1] ← SRC[15:8] ELSE (* Memory location unchanged *); FI;
(* Repeat operation for 3rd through 14th bytes in source operand *)
IF (MASK[127] = 1)
    THEN DEST[DS:DI/EDI/RDI+15] ← SRC[127:120] ELSE (* Memory location unchanged *); FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_maskmoveu_si128(__m128i d, __m128i n, char * p)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L= 1 If VEX.vvvv != 1111B

VMASKMOV- Conditional SIMD Packed Loads and Stores

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 2C /r VMASKMOVPS xmm1, xmm2, m128	A	V/V	AVX	Conditionally load packed single-precision values from m128 using mask in xmm2 and store in xmm1
VEX.NDS.256.66.0F38.W0 2C /r VMASKMOVPS ymm1, ymm2, m256	A	V/V	AVX	Conditionally load packed single-precision values from m256 using mask in ymm2 and store in ymm1
VEX.NDS.128.66.0F38.W0 2D /r VMASKMOVPD xmm1, xmm2, m128	A	V/V	AVX	Conditionally load packed double-precision values from m128 using mask in xmm2 and store in xmm1
VEX.NDS.256.66.0F38.W0 2D /r VMASKMOVPD ymm1, ymm2, m256	A	V/V	AVX	Conditionally load packed double-precision values from m256 using mask in ymm2 and store in ymm1
VEX.NDS.128.66.0F38.W0 2E /r VMASKMOVPS m128, xmm1, xmm2	B	V/V	AVX	Conditionally store packed single-precision values from xmm2 using mask in xmm1
VEX.NDS.256.66.0F38.W0 2E /r VMASKMOVPS m256, ymm1, ymm2	B	V/V	AVX	Conditionally store packed single-precision values from ymm2 using mask in ymm1
VEX.NDS.128.66.0F38.W0 2F /r VMASKMOVPD m128, xmm1, xmm2	B	V/V	AVX	Conditionally store packed double-precision values from xmm2 using mask in xmm1

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.W0 2F /r VMASKMOVPD m256, ymm1, ymm2	B	V/V	AVX	Conditionally store packed double-precision values from ymm2 using mask in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instruction. The destination operand is a memory address for the store form of these instructions. The other operands are both XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O and un-cached memory as the access and the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

Operation

VMASKMOVPS - 256-bit load

DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0

VMASKMOVPS -128-bit load

DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:97] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[255:128] ← 0

VMASKMOVPD - 256-bit load

DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0

VMASKMOVPD - 128-bit load

DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[255:128] ← 0

VMASKMOVPS - 256-bit store

IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
IF (SRC1[159]) DEST[159:128] ←SRC2[159:128]
IF (SRC1[191]) DEST[191:160] ← SRC2[191:160]
IF (SRC1[223]) DEST[223:192] ← SRC2[223:192]

IF (SRC1[255]) DEST[255:224] ← SRC2[255:224]

VMASKMOVPS - 128-bit store
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]

VMASKMOVPD - 256-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ←SRC2[127:64]
IF (SRC1[191]) DEST[191:128] ← SRC2[191:128]
IF (SRC1[255]) DEST[255:192] ← SRC2[255:192]

VMASKMOVPD - 128-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ←SRC2[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

__m256 _mm256_maskload_ps(float const *a, __m256i mask)

void _mm256_maskstore_ps(float *a, __m256i mask, __m256 b)

__m256d _mm256_maskload_pd(double *a, __m256i mask);

void _mm256_maskstore_pd(double *a, __m256i mask, __m256d b);

__m128 _mm256_maskload_ps(float const *a, __m128i mask)

void _mm256_maskstore_ps(float *a, __m128i mask, __m128 b)

__m128d _mm256_maskload_pd(double *a, __m128i mask);

void _mm256_maskstore_pd(double *a, __m128i mask, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6 (No AC# reported for any mask bit combinations); additionally

#UD	If VEX.W = 1

MAXPD- Maximum of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5F /r MAXPD xmm1, xmm2/m128	A	V/V	SSE2	Return the maximum double-precision floating-point values between xmm1 and xmm2/mem
VEX.NDS.128.66.0F.WIG 5F /r VMAXPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the maximum double-precision floating-point values between xmm2 and xmm3/mem
VEX.NDS.256.66.0F.WIG 5F /r VMAXPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum packed double-precision floating-point values between ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

MAX(SRC1, SRC2)

```
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
        ELSE DEST ← SRC2;
    FI;
}
```

VMAXPD (VEX.256 encoded version)

```
DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← MAX(SRC1[127:64], SRC2[127:64])
DEST[191:128] ← MAX(SRC1[191:128], SRC2[191:128])
DEST[255:192] ← MAX(SRC1[255:192], SRC2[255:192])
```

VMAXPD (VEX.128 encoded version)

```
DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← MAX(SRC1[127:64], SRC2[127:64])
DEST[255:128] ← 0
```

MAXPD (128-bit Legacy SSE version)

```
DEST[63:0] ← MAX(DEST[63:0], SRC[63:0])
DEST[127:64] ← MAX(DEST[127:64], SRC[127:64])
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMAXPD __m256d _mm256_max_pd (__m256d a, __m256d b);

(V)MAXPD __m128d _mm_max_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

See Exceptions Type 2

MAXPS- Maximum of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 5F /r MAXPS xmm1, xmm2/m128	A	V/V	SSE	Return the maximum single-precision floating-point values between xmm1 and xmm2/mem
VEX.NDS.128.0F.WIG 5F /r VMAXPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the maximum single-precision floating-point values between xmm2 and xmm3/mem
VEX.NDS.256.0F.WIG 5F /r VMAXPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum single double-precision floating-point values between ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

MAX(SRC1, SRC2)

```
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
        ELSE DEST ← SRC2;
    FI;
}
```

VMAXPS (VEX.256 encoded version)

```
DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] ← MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MAX(SRC1[127:96], SRC2[127:96])
DEST[159:128] ← MAX(SRC1[159:128], SRC2[159:128])
DEST[191:160] ← MAX(SRC1[191:160], SRC2[191:160])
DEST[223:192] ← MAX(SRC1[223:192], SRC2[223:192])
DEST[255:224] ← MAX(SRC1[255:224], SRC2[255:224])
```

VMAXPS (VEX.128 encoded version)

```
DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] ← MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MAX(SRC1[127:96], SRC2[127:96])
DEST[255:128] ← 0
```

MAXPS (128-bit Legacy SSE version)

```
DEST[31:0] ← MAX(DEST[31:0], SRC[31:0])
DEST[63:32] ← MAX(DEST[63:32], SRC[63:32])
DEST[95:64] ← MAX(DEST[95:64], SRC[95:64])
DEST[127:96] ← MAX(DEST[127:96], SRC[127:96])
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMAXPS __m256 _mm256_max_ps (__m256 a, __m256 b);

MAXPS __m128 _mm_max_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

See Exceptions Type 2

MAXSD- Return Maximum Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5F /r MAXSD xmm1, xmm2/m64	A	V/V	SSE2	Return the maximum scalar double-precision floating-point value between xmm2/mem64 and xmm1.
VEX.NDS.LIG.F2.0F.WIG 5F /r VMAXSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the maximum scalar double-precision floating-point value between xmm3/mem64 and xmm2.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low double-precision floating-point values in the first source operand and second the source operand, and returns the maximum value to the low quad-word of the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. When the second source operand is a memory operand, only 64 bits are accessed. The high quadword of the destination operand is copied from the same bits of first source operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN of either source operand be returned, the action of MAXSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

Operation

MAX(SRC1, SRC2)

```
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
        ELSE DEST ← SRC2;
    FI;
}
```

VMAXSD (VEX.128 encoded version)

```
DEST[63:0] ←MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ←SRC1[127:64]
DEST[255:128] ← 0
```

MAXSD (128-bit Legacy SSE version)

```
DEST[63:0] ←MAX(DEST[63:0], SRC[63:0])
DEST[255:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

MAXSD __m128d _mm_max_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

See Exceptions Type 3

MAXSS- Return Maximum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5F /r MAXSS xmm1, xmm2/m32	A	V/V	SSE	Return the maximum scalar single-precision floating-point value between xmm2/mem32 and xmm1.
VEX.NDS.LIG.F3.0F.WIG 5F /r VMAXSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Return the maximum scalar single-precision floating-point value between xmm3/mem32 and xmm2.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low single-precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low double-word of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN from either source operand be returned, the action of MAXSS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

Operation

MAX(SRC1, SRC2)

```
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
        ELSE DEST ← SRC2;
    FI;
}
```

VMAXSS (VEX.128 encoded version)

DEST[31:0] ←MAX(SRC1[31:0], SRC2[31:0])
DEST[127:32] ←SRC1[127:32]
DEST[255:128] ← 0

MAXSS (128-bit Legacy SSE version)

DEST[31:0] ←MAX(DEST[31:0], SRC[31:0])
DEST[255:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

__m128 _mm_max_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

See Exceptions Type 3

MINPD- Minimum of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5D /r MINPD xmm1, xmm2/m128	A	V/V	SSE2	Return the minimum double-precision floating-point values between xmm1 and xmm2/mem
VEX.NDS.128.66.0F.WIG 5D /r VMINPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the minimum double-precision floating-point values between xmm2 and xmm3/mem
VEX.NDS.256.66.0F.WIG 5D /r VMINPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum packed double-precision floating-point values between ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

MIN(SRC1, SRC2)

```
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
        ELSE DEST ← SRC2;
    FI;
}
```

VMINPD (VEX.256 encoded version)

```
DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← MIN(SRC1[127:64], SRC2[127:64])
DEST[191:128] ← MIN(SRC1[191:128], SRC2[191:128])
DEST[255:192] ← MIN(SRC1[255:192], SRC2[255:192])
```

VMINPD (VEX.128 encoded version)

```
DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← MIN(SRC1[127:64], SRC2[127:64])
DEST[255:128] ← 0
```

MINPD (128-bit Legacy SSE version)

```
DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← MIN(SRC1[127:64], SRC2[127:64])
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMINPD __m256d _mm256_min_pd (__m256d a, __m256d b);

MINPD __m128d _mm_min_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

See Exceptions Type 2

MINPS- Minimum of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 5D /r MINPS xmm1, xmm2/m128	A	V/V	SSE	Return the minimum single-precision floating-point values between xmm1 and xmm2/mem
VEX.NDS.128.0F.WIG 5D /r VMINPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the minimum single-precision floating-point values between xmm2 and xmm3/mem
VEX.NDS.256.0F.WIG 5D /r VMINPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum single double-precision floating-point values between ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

MIN(SRC1, SRC2)

```
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
        ELSE DEST ← SRC2;
    FI;
}
```

VMINPS (VEX.256 encoded version)

```
DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ← MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MIN(SRC1[127:96], SRC2[127:96])
DEST[159:128] ← MIN(SRC1[159:128], SRC2[159:128])
DEST[191:160] ← MIN(SRC1[191:160], SRC2[191:160])
DEST[223:192] ← MIN(SRC1[223:192], SRC2[223:192])
DEST[255:224] ← MIN(SRC1[255:224], SRC2[255:224])
```

VMINPS (VEX.128 encoded version)

```
DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ← MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MIN(SRC1[127:96], SRC2[127:96])
DEST[255:128] ← 0
```

MINPS (128-bit Legacy SSE version)

```
DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ← MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MIN(SRC1[127:96], SRC2[127:96])
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMINPS __m256 _mm256_min_ps (__m256 a, __m256 b);

MINPS __m128 _mm_min_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

See Exceptions Type 2

MINSD- Return Minimum Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5D /r MINSD xmm1, xmm2/m64	A	V/V	SSE2	Return the minimum scalar double precision floating-point value between xmm2/mem64 and xmm1.
VEX.NDS.LIG.F2.0F.WIG 5D /r VMINSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the minimum scalar double precision floating-point value between xmm3/mem64 and xmm2.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low double-precision floating-point values in the first source operand and the second source operand, and returns the minimum value to the low quadword of the destination operand. When the source operand is a memory operand, only the 64 bits are accessed. The high quadword of the destination operand is copied from the same bits in the first source operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second source) be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

Operation

MIN(SRC1, SRC2)

```
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
        ELSE DEST ← SRC2;
    FI;
}
```

MINSD (VEX.128 encoded version)

```
DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0
```

MINSD (128-bit Legacy SSE version)

```
DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

MINSD __m128d _mm_min_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

See Exceptions Type 3

MINSS- Return Minimum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5D /r MINSS xmm1,xmm2/m32	A	V/V	SSE	Return the minimum scalar single precision floating-point value between xmm2/mem32 and xmm1.
VEX.NDS.LIG.F3.0F.WIG 5D /r VMINSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Return the minimum scalar single precision floating-point value between xmm3/mem32 and xmm2.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low single-precision floating-point values in the first source operand and the second source operand and returns the minimum value to the low double-word of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN in either source operand be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

Operation

MIN(SRC1, SRC2)

```
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
        ELSE DEST ← SRC2;
    FI;
}
```

VMINSS (VEX.128 encoded version)

```
DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[127:32] ← SRC1[127:32]
DEST[255:128] ← 0
```

MINSS (128-bit Legacy SSE version)

```
DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

MINSS __m128 _mm_min_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

See Exceptions Type 3

MOVAPD- Move Aligned Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 28 /r MOVAPD xmm1, xmm2/m128	A	V/V	SSE2	Move aligned packed double-precision floating-point values from xmm2/mem to xmm1
66 0F 29 /r MOVAPD xmm2/m128, xmm1	B	V/V	SSE2	Move aligned packed double-precision floating-point values from xmm1 to xmm2/mem
VEX.128.66.0F.WIG 28 /r VMOVAPD xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed double-precision floating-point values from xmm2/mem to xmm1
VEX.128.66.0F.WIG 29 /r VMOVAPD xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed double-precision floating-point values from xmm1 to xmm2/mem
VEX.256.66.0F.WIG 28 /r VMOVAPD ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed double-precision floating-point values from ymm2/mem to ymm1
VEX.256.66.0F.WIG 29 /r VMOVAPD ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed double-precision floating-point values from ymm1 to ymm2/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves 2 or 4 double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM or YMM register from an 128-bit or 256-bit memory location, to store the contents of an XMM or YMM register into a 128-bit or 256-bit memory location, or to move data between two XMM or two YMM registers. When the source or destina-

tion operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary or a general-protection exception (#GP) will be generated. To move double-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

VEX.256 encoded version:

Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit versions:

Moves 128 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register destination are zeroed.

Operation

VMOVAPD (VEX.256 encoded version)

DEST[255:0] ← SRC[255:0]

VMOVAPD (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]
DEST[255:128] ← 0

MOVAPD (128-bit load- and register-copy- form Legacy SSE version)

DEST[127:0] ← SRC[127:0]
DEST[255:128] (Unmodified)

(V)MOVAPD (128-bit store-form version)

DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVAPD __m256d _mm256_load_pd (double const * p);

VMOVAPD _mm256_store_pd(double * p, __m256d a);

MOVAPD __m128d _mm_load_pd (double const * p);

MOVAPD _mm_store_pd(double * p, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type1.SSE2; additionally

#UD	If VEX.vvvv != 1111B.

MOVAPS- Move Aligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 28 /r MOVAPS xmm1, xmm2/m128	A	V/V	SSE	Move aligned packed single-precision floating-point values from xmm2/mem to xmm1
0F 29 /r MOVAPS xmm2/m128, xmm1	B	V/V	SSE	Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem
VEX.128.0F.WIG 28 /r VMOVAPS xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed single-precision floating-point values from xmm2/mem to xmm1
VEX.128.0F.WIG 29 /r VMOVAPS xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem
VEX.256.0F.WIG 28 /r VMOVAPS ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed single-precision floating-point values from ymm2/mem to ymm1
VEX.256.0F.WIG 29 /r VMOVAPS ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed single-precision floating-point values from ymm1 to ymm2/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves 4 or8 single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM or YMM register from an 128-bit or 256-bit memory location, to store the contents of an XMM or YMM register into a 128-bit or 256-bit memory location, or to move data between two XMM or two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-

bit version) or 32-byte (VEX.256 encoded version) boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

VEX.256 encoded version:

Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

128-bit versions:

Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VMOVAPS (VEX.256 encoded version)

DEST[255:0] ← SRC[255:0]

VMOVAPS (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]
DEST[255:128] ← 0

MOVAPS (128-bit load- and register-copy- form Legacy SSE version)

DEST[127:0] ← SRC[127:0]
DEST[255:128] (Unmodified)

(V)MOVAPS (128-bit store form)

DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVAPS __m256 _mm256_load_ps (float const * p);

VMOVAPS _mm256_store_ps(float * p, __m256 a);

MOVAPS __m128 _mm_load_ps (float const * p);

MOVAPS _mm_store_ps(float * p, __m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type1.SSE; additionally

#UD	If VEX.vvvv != 1111B.

MOVD/MOVQ- Move Doubleword and Quadword

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 6E /r MOVD xmm1, r32/m32	A	V/V	SSE2	Move doubleword from r/m32 to xmm1
66 REX.W 0F 6E /r MOVQ xmm1, r64/m64	A	V/N.E.	SSE2	Move quadword from r/m64 to xmm1
VEX.128.66.0F.W0 6E /r VMOVD xmm1, r32/m32	A	V/V	AVX	Move doubleword from r/m32 to xmm1
VEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64	A	V/N.E.	AVX	Move quadword from r/m64 to xmm1
66 0F 7E /r MOVD r32/m32, xmm1	B	V/V	SSE2	Move doubleword from xmm1 register to r/m32
66 REX.W 0F 7E /r MOVQ r64/m64, xmm1	B	V/N.E.	SSE2	Move quadword from xmm1 register to r/m64
VEX.128.66.0F.W0 7E /r VMOVD r32/m32, xmm1	B	V/V	AVX	Move doubleword from xmm1 register to r/m32
VEX.128.66.0F.W1 7E /r VMOVQ r64/m64, xmm1	B	V/N.E.	AVX	Move quadword from xmm1 register to r/m64

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

MOVD/Q with XMM destination:

Moves a dword integer from the source operand and stores it in the low 32-bits of the destination XMM register. The upper bits of the destination are zeroed. The source

operand can be a 32-bit register or 32-bit memory location. A REX.W prefix promotes this to copy qword integers.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Note: In 32-bit mode, the functionality of VMOVQ is not encodable, the instruction bytes that encode VMOVQ (VEX.W=1) is executed as VMOVD (treating VEX.W = 0 instead).

MOVD/Q with r32/m32 or r64/m64 destination:

Stores 32 (64) bits from the low bits of the source XMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

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

Operation

MOVD (Legacy SSE version when destination is an XMM register)
DEST[31:0] ← SRC[31:0]
DEST[127:32] ← 0H
DEST[255:128] (Unmodified)

VMOVD (VEX-encoded version when destination is an XMM register)
DEST[31:0] ← SRC[31:0]
DEST[255:32] ← 0H

MOVQ (Legacy SSE version when destination is an XMM register)
DEST[63:0] ← SRC[63:0]
DEST[127:64] ← 0H
DEST[255:128] (Unmodified)

VMOVQ (VEX-encoded version when destination is an XMM register)
DEST[63:0] ← SRC[63:0]
DEST[255:64] ← 0H

MOVD / VMOVD (when destination is not an XMM register)
DEST[31:0] ← SRC[31:0]

MOVQ / VMOVQ (when destination is not an XMM register)
DEST[63:0] ← SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVD __m128i _mm_cvtsi32_si128(int a)

MOVD int _mm_cvtsi128_si32(__m128i a)

MOVQ __m128i _mm_cvtsi64_si128(__int64 a)

MOVQ __int64 _mm_cvtsi128_si64(__m128i a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

MOVQ- Move Quadword

Opcode Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 7E /r MOVQ xmm1, xmm2/m64	A	V/V	SSE2	Move quadword from xmm2/m64 to xmm1
VEX.128.F3.0F.WIG 7E /r VMOVQ xmm1, xmm2	A	V/V	AVX	Move quadword from xmm2 to xmm1
VEX.128.F3.0F.WIG 7E /r VMOVQ xmm1, m64	A	V/V	AVX	Load quadword from m64 to xmm1
66 0F D6 /r MOVQ xmm1/m64, xmm2	B	V/V	SSE2	Move quadword from xmm2 register to xmm1/m64
VEX.128.66.0F.WIG D6 /r VMOVQ xmm1/m64, xmm2	B	V/V	AVX	Move quadword from xmm2 register to xmm1/m64

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Copies a quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be an XMM register or a 64-bit memory locations. This instruction can be used to move data between two XMM registers or between an XMM register and a 64-bit memory location. The instruction cannot be used to transfer data between memory locations.

When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared to all 0s.

Note: In VEX.128.66.0F D6 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note: In VEX.128.F3.0F 7E version, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

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

Operation

MOVQ (F3 0F 7E and 66 0F D6) with XMM register source and destination:
DEST[63:0] ← SRC[63:0]
DEST[127:64] ← 0
DEST[255:128] (Unmodified)

VMOVQ (VEX.NDS.128.F3.0F 7E) with XMM register source and destination:
DEST[63:0] ← SRC[63:0]
DEST[255:64] ← 0

VMOVQ (VEX.128.66.0F D6) with XMM register source and destination:
DEST[63:0] ← SRC[63:0]
DEST[255:64] ← 0

MOVQ (7E) with memory source:
DEST[63:0] ← SRC[63:0]
DEST[127:64] ← 0000000000000000H
DEST[255:128] (Unmodified)

VMOVQ (7E) with memory source:
DEST[63:0] ← SRC[63:0]
DEST[255:64] ← 0000000000000000H

MOVQ (D6) with memory dest:
DEST[63:0] ← SRC[63:0]

VMOVQ (D6) with memory dest:
DEST[63:0] ← SRC2[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVQ __m128i_mm_move_epi64(__m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.

If VEX.vvvv != 1111B.

MOVDDUP- Replicate Double FP Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 12 /r MOVDDUP xmm1, xmm2/m64	A	V/V	SSE3	Move double-precision floating-point values from xmm2/mem and duplicate into xmm1
VEX.128.F2.0F.WIG 12 /r VMOVDDUP xmm1, xmm2/m64	A	V/V	AVX	Move double-precision floating-point values from xmm2/mem and duplicate into xmm1
VEX.256.F2.0F.WIG 12 /r VMOVDDUP ymm1, ymm2/m256	A	V/V	AVX	Move even index double-precision floating-point values from ymm2/mem and duplicate each element into ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

VEX.256 encoded version:

Duplicates even-indexed double-precision floating-point values from the source operand (second operand).

128-bit versions:

Duplicates a single double-precision floating-point value into the destination.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

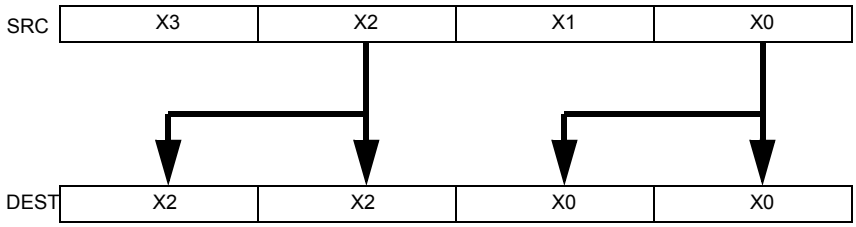


Figure 5-14. VMOVDDUP Operation

Operation

VMOVDDUP (VEX.256 encoded version)

DEST[63:0] ← SRC[63:0]
DEST[127:64] ← SRC[63:0]
DEST[191:128] ← SRC[191:128]
DEST[255:192] ← SRC[191:128]

VMOVDDUP (VEX.128 encoded version)

DEST[63:0] ← SRC[63:0]
DEST[127:64] ← SRC[63:0]
DEST[255:128] ← 0

MOVDDUP (128-bit Legacy SSE version)

DEST[63:0] ← SRC[63:0]
DEST[127:64] ← SRC[63:0]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

MOVDDUP __m256d _mm256_movedup_pd (__m256d a);

MOVDDUP __m128d _mm_movedup_pd (__m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.vvvv != 1111B.

MOVDQA- Move Aligned Packed Integer Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 6F /r MOVDQA xmm1, xmm2/m128	A	V/V	SSE2	Move aligned packed integer values from xmm2/mem to xmm1
66 0F 7F /r MOVDQA xmm2/m128, xmm1	B	V/V	SSE2	Move aligned packed integer values from xmm1 to xmm2/mem
VEX.128.66.0F.WIG 6F /r VMOVDQA xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed integer values from xmm2/mem to xmm1
VEX.128.66.0F.WIG 7F /r VMOVDQA xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed integer values from xmm1 to xmm2/mem
VEX.256.66.0F.WIG 6F /r VMOVDQA ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed integer values from ymm2/mem to ymm1
VEX.256.66.0F.WIG 7F /r VMOVDQA ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed integer values from ymm1 to ymm2/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VMOVDQA (VEX.256 encoded version)
DEST[255:0] ← SRC[255:0]

VMOVDQA (VEX.128 encoded version)
DEST[127:0] ← SRC[127:0]
DEST[255:128] ← 0

MOVDQA (128-bit load- and register- form Legacy SSE version)
DEST[127:0] ← SRC[127:0]
DEST[255:128] (Unmodified)

(V)MOVDQA (128-bit store forms)
DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVDQA __m256i _mm256_load_si256 (__m256i * p);

VMOVDQA _mm256_store_si256(_m256i *p, __m256i a);

MOVDQA __m128i _mm_load_si128 (__m128i * p);

MOVDQA _mm_store_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type1.SSE2; additionally

#UD	If VEX.vvvv != 1111B.

MOVDQU- Move Unaligned Packed Integer Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 6F /r MOVDQU xmm1, xmm2/m128	A	V/V	SSE2	Move unaligned packed integer values from xmm2/mem to xmm1
F3 0F 7F /r MOVDQU xmm2/m128, xmm1	B	V/V	SSE2	Move unaligned packed integer values from xmm1 to xmm2/mem
VEX.128.F3.0F.WIG 6F /r VMOVDQU xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed integer values from xmm2/mem to xmm1
VEX.128.F3.0F.WIG 7F /r VMOVDQU xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed integer values from xmm1 to xmm2/mem
VEX.256.F3.0F.WIG 6F /r VMOVDQU ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed integer values from ymm2/mem to ymm1
VEX.256.F3.0F.WIG 7F /r VMOVDQU ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed integer values from ymm1 to ymm2/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned to any alignment without causing a general-protection exception (#GP) to be generated

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VMOVDQU (VEX.256 encoded version)
DEST[255:0] ← SRC[255:0]

VMOVDQU (VEX.128 encoded version)
DEST[127:0] ← SRC[127:0]
DEST[255:128] ← 0

MOVDQU load and register copy (128-bit Legacy SSE version)
DEST[127:0] ← SRC[127:0]
DEST[255:128] (Unmodified)

(V)MOVDQU 128-bit store-form versions
DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVDQU __m256i _mm256_loadu_si256 (__m256i * p);

VMOVDQU _mm256_storeu_si256(_m256i *p, __m256i a);

MOVDQU __m128i _mm_loadu_si128 (__m128i * p);

MOVDQU _mm_storeu_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv != 1111B.

MOVHLPS - Move Packed Single-Precision Floating-Point Values High to Low

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 12 /r MOVHLPS xmm1, xmm2	A	V/V	SSE	Move two packed single-precision floating-point values from high quadword of xmm2 to low quadword of xmm1.
VEX.NDS.128.0F.WIG 12 /r VMOVHLPS xmm1, xmm2, xmm3	B	V/V	AVX	Merge two packed single-precision floating-point values from high quadword of xmm3 and low quadword of xmm2.

Instruction Operand Encoding[1]

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single-precision floating-point values from the high quadword of the second XMM argument (second operand) to the low quadword of the first XMM register (first argument). The high quadword of the destination operand is left unchanged. The upper 128 bits of the corresponding YMM destination register are unmodified.

128-bit three-argument form

Moves two packed single-precision floating-point values from the high quadword of the third XMM argument (third operand) to the low quadword of the destination (first operand). Copies the high quadword from the second XMM argument (second operand) to the high quadword of the destination (first operand). The upper 128-bits of the destination YMM register are zeroed.

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

1. ModRM.MOD = 011B required

Operation

MOVHLPS (128-bit two-argument form)

DEST[63:0] ← SRC[127:64]
DEST[255:64] (Unmodified)

VMOVHLPS (128-bit three-argument form)

DEST[63:0] ← SRC2[127:64]
DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVHLPS __m128 _mm_movehl_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.L = 1

MOVHPD- Move High Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 16 /r MOVHPD xmm1, m64	A	V/V	SSE2	Move double-precision floating-point values from m64 to high quadword of xmm1.
VEX.NDS.128.66.0F.WIG 16 /r VMOVHPD xmm2, xmm1, m64	B	V/V	AVX	Merge double-precision floating-point value from m64 and the low quadword of xmm1.
66 0F 17/r MOVHPD m64, xmm1	C	V/V	SSE2	Move double-precision floating-point values from high quadword of xmm1 to m64.
VEX128.66.0F.WIG 17/r VMOVHPD m64, xmm1	C	V/V	AVX	Move double-precision floating-point values from high quadword of xmm1 to m64.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

VEX.128 encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from second XMM register (second operand)

are stored in the lower 64-bits of the destination. The upper 128-bits of the destination YMM register are zeroed.

128-bit store:

Stores a double-precision floating-point value from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPD (store) (VEX.128.66.0F 17 /r) is legal and has the same behavior as the existing 66 0F 17 store. For VMOVHPD (store) (VEX.128.66.0F 17 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

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

Operation

MOVHPD (128-bit Legacy SSE load)

```
DEST[63:0] (Unmodified)
DEST[127:64] ← SRC[63:0]
DEST[255:128] (Unmodified)
```

VMOVHPD (VEX.128 encoded load)

```
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[255:128] ← 0
```

VMOVHPD (store)

```
DEST[63:0] ← SRC[127:64]
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPD __m128d _mm_loadh_pd (__m128d a, double *p)

MOVHPD void _mm_storeh_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.

MOVHPS- Move High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 16 /r MOVHPS xmm1, m64	A	V/V	SSE	Move two packed single-precision floating-point values from m64 to high quadword of xmm1.
VEX.NDS.128.0F.WIG 16 /r VMOVHPS xmm2, xmm1, m64	B	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.
0F 17/r MOVHPS m64, xmm1	C	V/V	SSE	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.
VEX.128.0F.WIG 17/r VMOVHPS m64, xmm1	C	V/V	AVX	Move two packed single-precision floating-point values from high quadword of xmm1to m64.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

VEX.128 encoded load:

Loads two single-precision floating-point values from the source 64-bit memory operand (third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from second XMM register (second operand)

are stored in the lower 64-bits of the destination. The upper 128-bits of the destination YMM register are zeroed.

128-bit store:

Stores two packed single-precision floating-point values from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPS (store) (VEX.NDS.128.0F 17 /r) is legal and has the same behavior as the existing 0F 17 store. For VMOVHPS (store) (VEX.NDS.128.0F 17 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

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

Operation

MOVHPS (128-bit Legacy SSE load)

```
DEST[63:0] (Unmodified)
DEST[127:64] ← SRC[63:0]
DEST[255:128] (Unmodified)
```

VMOVHPS (VEX.128 encoded load)

```
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[255:128] ← 0
```

VMOVHPS (store)

```
DEST[63:0] ← SRC[127:64]
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPS __m128d _mm_loadh_pi (__m128d a, __m64 *p)

MOVHPS void _mm_storeh_pi (__m64 *p, __m128d a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.

MOVLHPS - Move Packed Single-Precision Floating-Point Values Low to High

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 16 /r MOVLHPS xmm1, xmm2		V/V	SSE	Move two packed single-precision floating-point values from low quadword of xmm2 to high quadword of xmm1.
VEX.NDS.128.0F.WIG 16 /r VMOVLHPS xmm1, xmm2, xmm3		V/V	AVX	Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2.

Instruction Operand Encoding[1]

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single-precision floating-point values from the low quadword of the second XMM argument (second operand) to the high quadword of the first XMM register (first argument). The low quadword of the destination operand is left unchanged. The upper 128 bits of the corresponding YMM destination register are unmodified.

128-bit three-argument form

Moves two packed single-precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). The upper 128-bits of the destination YMM register are zeroed.

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

1. ModRM.MOD = 011B required

Operation

MOVLHPS (128-bit two-argument form)

DEST[63:0] (Unmodified)
DEST[127:64] ← SRC[63:0]
DEST[255:128] (Unmodified)

VMOVLHPS (128-bit three-argument form)

DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVLHPS __m128 _mm_movelh_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

#UD	IF VEX.L = 1.

MOVLPD- Move Low Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 12 /r MOVLPD xmm1, m64	A	V/V	SSE2	Move double-precision floating-point values from m64 to low quadword of xmm1.
VEX.NDS.128.66.0F.WIG 12 /r VMOVLPD xmm2, xmm1, m64	B	V/V	AVX	Merge double-precision floating-point value from m64 and the high quadword of xmm1.
66 0F 13/r MOVLPD m64, xmm1	C	V/V	SSE2	Move double-precision floating-point values from low quadword of xmm1 to m64.
VEX.128.66.0F.WIG 13/r VMOVLPD m64, xmm1	C	V/V	AVX	Move double-precision floating-point values from low quadword of xmm1 to m64.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (r)	ModRM:reg (r, w)	VEX.vvvv (r)	NA
C	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

VEX.128 encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (third operand), merges it with the upper 64-bits of the first source XMM register (second operand), and stores it in the low 128-bits of the destination XMM

register (first operand). The upper 128-bits of the destination YMM register are zeroed.

128-bit store:

Stores a double-precision floating-point value from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPD (store) (VEX.128.66.0F 13 /r) is legal and has the same behavior as the existing 66 0F 13 store. For VMOVLPD (store) (VEX.128.66.0F 13 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

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

Operation

MOVLPD (128-bit Legacy SSE load)

DEST[63:0] ← SRC[63:0]
DEST[255:64] (Unmodified)

VMOVLPD (VEX.128 encoded load)

DEST[63:0] ← SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0

VMOVLPD (store)

DEST[63:0] ← SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPD __m128d _mm_loadl_pd (__m128d a, double *p)
MOVLPD void _mm_storel_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

MOVLPS- Move Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 12 /r MOVLPS xmm1, m64	A	V/V	SSE	Move two packed single-precision floating-point values from m64 to low quadword of xmm1.
VEX.NDS.128.0F.WIG 12 /r VMOVLPS xmm2, xmm1, m64	B	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.
0F 13/r MOVLPS m64, xmm1	C	V/V	SSE	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.
VEX.128.0F.WIG 13/r VMOVLPS m64, xmm1	C	V/V	AVX	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

VEX.128 encoded load:

Loads two packed single-precision floating-point values from the source 64-bit memory operand (third operand), merges them with the upper 64-bits of the first source XMM register (second operand), and stores them in the low 128-bits of the

destination XMM register (first operand). The upper 128-bits of the destination YMM register are zeroed.

128-bit store:

Loads two packed single-precision floating-point values from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPS (store) (VEX.128.0F 13 /r) is legal and has the same behavior as the existing 0F 13 store. For VMOVLPS (store) (VEX.128.0F 13 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

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

Operation

MOVLPS (128-bit Legacy SSE load)

DEST[63:0] ← SRC[63:0]
DEST[255:64] (Unmodified)

VMOVLPS (VEX.128 encoded load)

DEST[63:0] ← SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0

VMOVLPS (store)

DEST[63:0] ← SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPS __m128 _mm_loadl_pi (__m128 a, __m64 *p)
MOVLPS void _mm_storel_pi (__m64 *p, __m128 a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

MOVMSKPD- Extract Double-Precision Floating-Point Sign mask

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 50 /r MOVMSKPD reg, xmm2	A	V/V	SSE2	Extract 2-bit sign mask from xmm2 and store in reg. The upper bits of r32 or r64 are zero'ed.
VEX.128.66.0F.WIG 50 /r VMOVMSKPD reg, xmm2	A	V/V	AVX	Extract 2-bit sign mask from xmm2 and store in reg. The upper bits of r32 or r64 are zero'ed.
VEX.256.66.0F.WIG 50 /r VMOVMSKPD reg, ymm2	A	V/V	AVX	Extract 4-bit sign mask from ymm2 and store in reg. The upper bits of r32 or r64 are zero'ed.

Instruction Operand Encoding[1]

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Extracts the sign bits from the packed double-precision floating-point values in the source operand (second operand), formats them into a 2- or 4-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM or YMM register, and the destination operand is a general-purpose register. The mask is stored in the 2 or 4 low-order bits of the destination operand. The upper bits of the destination operand beyond the mask are filled with zeros.

In 64-bit mode, the default operand size of the destination register is 64 bit.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VMOVMSKPD (VEX.256 encoded version)

1. ModRM.MOD = 011B required

```
DEST[0] ← SRC[63]
DEST[1] ← SRC[127]
DEST[2] ← SRC[191]
DEST[3] ← SRC[255]
IF DEST = r32
    THEN DEST[31:4] ← 0;
    ELSE DEST[63:4] ← 0;
FI
```

(V)MOVMSKPD (128-bit versions)

```
DEST[0] ← SRC[63]
DEST[1] ← SRC[127]
IF DEST = r32
    THEN DEST[31:2] ← 0;
    ELSE DEST[63:2] ← 0;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

int _mm256_movemask_pd(__m256d a)

int _mm_movemask_pd(__m128d a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.vvvv != 1111B.

MOVMSKPS- Extract Single-Precision Floating-Point Sign mask

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 50 /r MOVMSKPS reg, xmm2	A	V/V	SSE	Extract 4-bit sign mask from xmm2 and store in reg. The upper bits of r32 or r64 are zero'ed.
VEX.128.0F.WIG 50 /r VMOVMSKPS reg, xmm2	A	V/V	AVX	Extract 4-bit sign mask from xmm2 and store in reg. The upper bits of r32 or r64 are zero'ed.
VEX.256.0F.WIG 50 /r VMOVMSKPS reg, ymm2	A	V/V	AVX	Extract 8-bit sign mask from ymm2 and store in reg. The upper bits of r32 or r64 are zero'ed.

Instruction Operand Encoding[1]

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Extracts the sign bits from the packed single-precision floating-point values in the source operand (second operand), formats them into a 4- or 8-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM or YMM register, and the destination operand is a general-purpose register. The mask is stored in the 4 or 8 low-order bits of the destination operand. The upper bits of the destination operand beyond the mask are filled with zeros.

In 64-bit mode, the default operand size of the destination register is 64 bit.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VMOVMSKPS (VEX.256 encoded version)

1. ModRM.MOD = 011B required

```
DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
DEST[4] ← SRC[159]
DEST[5] ← SRC[191]
DEST[6] ← SRC[223]
DEST[7] ← SRC[255]
IF DEST = r32
    THEN DEST[31:8] ← 0;
    ELSE DEST[63:8] ← 0;
FI
```

(V)MOVMSKPS (128-bit version)

```
DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
IF DEST = r32
    THEN DEST[31:4] ← 0;
    ELSE DEST[63:4] ← 0;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

int _mm256_movemask_ps(__m256 a)

int _mm_movemask_ps(__m128 a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.vvvv != 1111B.

MOVNTDQ- Store Packed Integers Using Non-Temporal Hint

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E7 /r MOVNTDQ m128, xmm1	A	V/V	SSE2	Move packed integer values in xmm1 to m128 using non-temporal hint
VEX.128.66.0F.WIG E7 /r VMOVNTDQ m128, xmm1	A	V/V	AVX	Move packed integer values in xmm1 to m128 using non-temporal hint
VEX.256.66.0F.WIG E7 /r VMOVNTDQ m256, ymm1	A	V/V	AVX	Move packed integer values in ymm1 to m256 using non-temporal hint

Instruction Operand Encoding[1]

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the packed integers in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register or YMM register, which is assumed to contain integer data (packed bytes, words, doublewords, or quadwords). The destination operand is a 128-bit or 256-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1.

1. ModRM.MOD = 011B required

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with VMOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

MOVNTDQ

DEST ← SRC

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTDQ void _mm256_stream_si256 (__m256i * p, __m256i a);

MOVNTDQ void _mm_stream_si128 (__m128i * p, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type1.SSE2; additionally

#UD	If VEX.vvvv != 1111B.

MOVNTDQA- Load Double Quadword Non-Temporal Aligned Hint

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2A /r MOVNTDQA xmm1, m128	A	V/V	SSE4_1	Move double quadword from m128 to xmm1 using non-temporal hint if WC memory type.
VEX.128.66.0F38.WIG 2A /r VMOVNTDQA xmm1, m128	A	V/V	AVX	Move double quadword from m128 to xmm using non-temporal hint if WC memory type.

Instruction Operand Encoding[1]

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

MOVNTDQA loads a double quadword from the source operand (second operand) to the destination operand (first operand) using a non-temporal hint if the memory source is WC (write combining) memory type. For WC memory type, the nontemporal hint may be implemented by loading a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Any memory-type aliased lines in the cache will be snooped and flushed. Subsequent MOVNTDQA reads to unread portions of the WC cache line will receive data from the temporary internal buffer if data is available. The temporary internal buffer may be flushed by the processor at any time for any reason, for example:

• A load operation other than a MOVNTDQA which references memory already resident in a temporary internal buffer.

• A non-WC reference to memory already resident in a temporary internal buffer.

• Interleaving of reads and writes to a single temporary internal buffer.

• Repeated (V)MOVNTDQA loads of a particular 16-byte item in a streaming line.

• Certain micro-architectural conditions including resource shortages, detection of a mis-speculation condition, and various fault conditions

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when reading the data from memory. Using this protocol, the processor

1. ModRM.MOD = 011B required

does not read the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being read can override the non-temporal hint, if the memory address specified for the non-temporal read is not a WC memory region. Information on non-temporal reads and writes can be found in "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the Intel® 64 and IA-32 Architecture Software Developer's Manual, Volume 3A.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with a MFENCE instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might use different memory types for the referenced memory locations or to synchronize reads of a processor with writes by other agents in the system. A processor's implementation of the streaming load hint does not override the effective memory type, but the implementation of the hint is processor dependent. For example, a processor implementation may choose to ignore the hint and process the instruction as a normal MOVDQA for any memory type. Alternatively, another implementation may optimize cache reads generated by MOVNTDQA on WB memory type to reduce cache evictions.

The 128-bit (V)MOVNTDQA addresses must be 16-byte aligned or the instruction will cause a #GP.

Note: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

MOVNTDQA (128bit- Legacy SSE form)

DEST ← SRC
DEST[255:128] (Unmodified)

VMOVNTDQA (VEX.128 encoded form)

DEST ← SRC
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTDQA __m128i _mm_stream_load_si128 (__m128i *p);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type1.SSE4.1; additionally

#UD	If VEX.vvvv != 1111B.
	If VEX.L = 1.

MOVNTPD- Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2B /r MOVNTPD m128, xmm1	A	V/V	SSE2	Move packed double-precision values in xmm1 to m128 using non-temporal hint
VEX.128.66.0F.WIG 2B /r VMOVNTPD m128, xmm1	A	V/V	AVX	Move packed double-precision values in xmm1 to m128 using non-temporal hint
VEX.256.66.0F.WIG 2B /r VMOVNTPD m256, ymm1	A	V/V	AVX	Move packed double-precision values in ymm1 to m256 using non-temporal hint

Instruction Operand Encoding[1]

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the packed double-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register or YMM register, which is assumed to contain packed double-precision, floating-pointing data. The destination operand is a 128-bit or 256-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of

1. ModRM.MOD = 011B required

Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

MOVNTPD

DEST ← SRC

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTPD void _mm256_stream_pd (double * p, __m256d a);

MOVNTPD void _mm_stream_pd (double * p, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type1.SSE2; additionally

#UD	If VEX.vvvv != 1111B.

MOVNTPS- Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 2B /r MOVNTPS m128, xmm1	A	V/V	SSE	Move packed single-precision values xmm1 to mem using non-temporal hint
VEX.128.0F.WIG 2B /r VMOVNTPS m128, xmm1	A	V/V	AVX	Move packed single-precision values xmm1 to mem using non-temporal hint
VEX.256.0F.WIG 2B /r VMOVNTPS m256, ymm1	A	V/V	AVX	Move packed single-precision values ymm1 to mem using non-temporal hint

Instruction Operand Encoding[1]

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the packed single-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register or YMM register, which is assumed to contain packed single-precision, floating-pointing. The destination operand is a 128-bit or 256-bitmemory location. The memory operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of

1. ModRM.MOD = 011B required

Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

MOVNTPS

DEST ← SRC

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTPS void _mm_stream_ps (float * p, __m128d a);

VMOVNTPS void _mm256_stream_ps (float * p, __m256 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type1.SSE; additionally

#UD	If VEX.vvvv != 1111B.

MOVSD- Move or Merge Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 10 /r MOVSD xmm1, xmm2	A	V/V	SSE2	Merge or Move scalar double-precision floating-point value from xmm2 to xmm1 register
F2 0F 10 /r MOVSD xmm1, m64	A	V/V	SSE2	Merge or Move scalar double-precision floating-point value from m64 to xmm1 register
VEX.NDS.LIG.F2.0F.WIG 10 /r VMOVSD xmm1, xmm2, xmm3	B	V/V	AVX	Merge scalar double-precision floating-point value from xmm2 and xmm3 to xmm1 register
VEX.LIG.F2.0F.WIG 10 /r VMOVSD xmm1, m64	D	V/V	AVX	Load scalar double-precision floating-point value from m64 to xmm1 register
F2 0F 11 /r MOVSD xmm2/m64, xmm1	C	V/V	SSE2	Move scalar double-precision floating-point value from xmm1 register to xmm2/m64
VEX.NDS.LIG.F2.0F.WIG 11 /r VMOVSD xmm1, xmm2, xmm3	E	V/V	AVX	Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1
VEX.LIG.F2.0F.WIG 11 /r VMOVSD m64, xmm1	C	V/V	AVX	Move scalar double-precision floating-point value from xmm1 register to m64

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	NA

Description

MOVSD moves a scalar double-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 64-bit memory locations. This instruc-

tion can be used to move a double-precision floating-point value to and from the low quadword of an XMM register and a 64-bit memory location, or to move a double-precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

For non-VEX encoded instruction syntax and when the source and destination operands are XMM registers, the high quadword of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the high quadword of the destination operand is cleared to all 0s.

Note: For the "VMOVSD m64, xmm1" (memory store form) instruction version, VEX.vvvv is reserved and must be 1111b, otherwise instruction will #UD.

Note: For the "VMOVSD xmm1, m64" (memory load form) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

VEX encoded instruction syntax supports two source operands and a destination operand if ModR/M.mod field is 11B. VEX.vvvv is used to encode the first source operand (the second operand). The low 128 bits of the destination operand stores the result of merging the low quadword of the second source operand with the quad word in bits 127:64 of the first source operand. The upper bits of the destination operand are cleared.

Operation

MOVSD (128-bit Legacy SSE version: MOVSD XMM1, XMM2)
DEST[63:0] ← SRC[63:0]
DEST[255:64] (Unmodified)

VMOVSD (VEX.NDS.128.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)
DEST[63:0] ← SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0

VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)
DEST[63:0] ← SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0

VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, m64)
DEST[63:0] ← SRC[63:0]
DEST[255:64] ← 0

MOVSD/VMOVSD (128-bit versions: MOVSD m64, xmm1 or VMOVSD m64, xmm1)
DEST[63:0] ← SRC[63:0]

MOVSD (128-bit Legacy SSE version: MOVSD XMM1, m64)
DEST[63:0] ← SRC[63:0]

DEST[127:64] ← 0
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

MOVSD __m128d _mm_load_sd (double *p)
MOVSD void _mm_store_sd (double *p, __m128d a)
MOVSD __m128d _mm_move_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.vvvv != 1111B.

MOVSHDUP- Replicate Single FP Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 16 /r MOVSHDUP xmm1, xmm2/m128	A	V/V	SSE3	Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1
VEX.128.F3.0F.WIG 16 /r VMOVSHDUP xmm1, xmm2/m128	A	V/V	AVX	Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1
VEX.256.F3.0F.WIG 16 /r VMOVSHDUP ymm1, ymm2/m256	A	V/V	AVX	Move odd index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Duplicates odd-indexed single-precision floating-point values from the source operand (second operand). See Figure 5-15. The source operand is an XMM or YMM register or 128 or 256-bit memory location and the destination operand is an XMM or YMM register.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

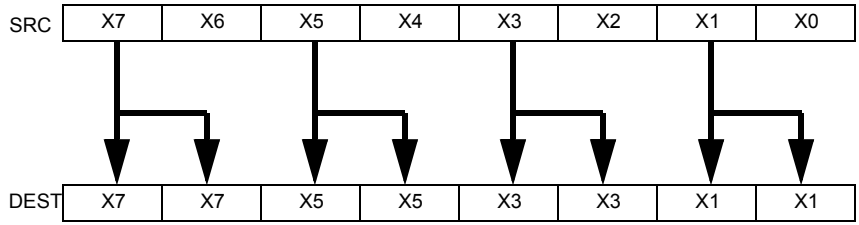


Figure 5-15. MOVSHDUP Operation

Operation

VMOVSHDUP (VEX.256 encoded version)

DEST[31:0] ← SRC[63:32]
DEST[63:32] ← SRC[63:32]
DEST[95:64] ← SRC[127:96]
DEST[127:96] ← SRC[127:96]
DEST[159:128] ← SRC[191:160]
DEST[191:160] ← SRC[191:160]
DEST[223:192] ← SRC[255:224]
DEST[255:224] ← SRC[255:224]

VMOVSHDUP (VEX.128 encoded version)

DEST[31:0] ← SRC[63:32]
DEST[63:32] ← SRC[63:32]
DEST[95:64] ← SRC[127:96]
DEST[127:96] ← SRC[127:96]
DEST[255:128] ← 0

MOVSHDUP (128-bit Legacy SSE version)

DEST[31:0] ← SRC[63:32]
DEST[63:32] ← SRC[63:32]
DEST[95:64] ← SRC[127:96]
DEST[127:96] ← SRC[127:96]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMOVSHDUP __m256 _mm256_movehdup_ps (__m256 a);
VMOVSHDUP __m128 _mm_movehdup_ps (__m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 2

MOVSLDUP- Replicate Single FP Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 12 /r MOVSLDUP xmm1, xmm2/m128	A	V/V	SSE3	Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1
VEX.128.F3.0F.WIG 12 /r VMOVSLDUP xmm1, xmm2/m128	A	V/V	AVX	Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1
VEX.256.F3.0F.WIG 12 /r VMOVSLDUP ymm1, ymm2/m256	A	V/V	AVX	Move even index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Duplicates even-indexed single-precision floating-point values from the source operand (second operand). See Figure 5-16. The source operand is an XMM or YMM register or 128 or 256-bit memory location and the destination operand is an XMM or YMM register.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

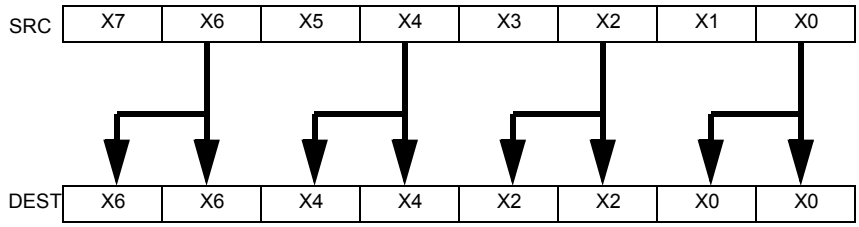


Figure 5-16. MOVSLDUP Operation

Operation

VMOVSLDUP (VEX.256 encoded version)

DEST[31:0] ← SRC[31:0]
DEST[63:32] ← SRC[31:0]
DEST[95:64] ← SRC[95:64]
DEST[127:96] ← SRC[95:64]
DEST[159:128] ← SRC[159:128]
DEST[191:160] ← SRC[159:128]
DEST[223:192] ← SRC[223:192]
DEST[255:224] ← SRC[223:192]

VMOVSLDUP (VEX.128 encoded version)

DEST[31:0] ← SRC[31:0]
DEST[63:32] ← SRC[31:0]
DEST[95:64] ← SRC[95:64]
DEST[127:96] ← SRC[95:64]
DEST[255:128] ← 0

MOVSLDUP (128-bit Legacy SSE version)

DEST[31:0] ← SRC[31:0]
DEST[63:32] ← SRC[31:0]
DEST[95:64] ← SRC[95:64]
DEST[127:96] ← SRC[95:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMOVSLDUP __m256 _mm256_moveldup_ps (__m256 a);
VMOVSLDUP __m128 _mm_moveldup_ps (__m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv != 1111B.

MOVSS- Move or Merge Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 10 /r MOVSS xmm1, xmm2	A	V/V	SSE	Merge scalar single-precision floating-point value from xmm2 to xmm1 register
F3 0F 10 /r MOVSS xmm1, m32	A	V/V	SSE	Load scalar single-precision floating-point value from m32 to xmm1 register
VEX.NDS.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, xmm2, xmm3	B	V/V	AVX	Merge scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register
VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, m32	D	V/V	AVX	Load scalar single-precision floating-point value from m32 to xmm1 register
F3 0F 11 /r MOVSS xmm2/m32, xmm1	C	V/V	SSE	Move scalar single-precision floating-point value from xmm1 register to xmm2/m32
VEX.NDS.LIG.F3.0F.WIG 11 /r VMOVSS xmm1, xmm2, xmm3	E	V/V	AVX	Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register
VEX.LIG.F3.0F.WIG 11 /r VMOVSS m32, xmm1	C	V/V	AVX	Move scalar single-precision floating-point value from xmm1 register to m32

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:reg (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	NA

Description

MOVSS moves a scalar single-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can

be used to move a single-precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single-precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

For non-VEX encoded syntax and when the source and destination operands are XMM registers, the high doublewords of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the high doublewords of the destination operand is cleared to all 0s.

Note: For the "VMOVSS m32, xmm1" (memory store form) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Note: For the "VMOVSS xmm1, m32" (memory load form) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

VEX encoded instruction syntax supports two source operands and a destination operand if ModR/M.mod field is 11B. VEX.vvvv is used to encode the first source operand (the second operand). The low 128 bits of the destination operand stores the result of merging the low dword of the second source operand with three dwords in bits 127:32 of the first source operand. The upper bits of the destination operand are cleared.

Operation

MOVSS (Legacy SSE version when the source and destination operands are both XMM registers)
DEST[31:0] ← SRC[31:0]
DEST[255:32] (Unmodified)

VMOVSS (VEX.NDS.128.F3.0F 11 /r where the destination is an XMM register)
DEST[31:0] ← SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[255:128] ← 0

VMOVSS (VEX.NDS.128.F3.0F 10 /r where the source and destination are XMM registers)
DEST[31:0] ← SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[255:128] ← 0

VMOVSS (VEX.NDS.128.F3.0F 10 /r when the source operand is memory and the destination is an XMM register)
DEST[31:0] ← SRC[31:0]
DEST[255:32] ← 0

MOVSS/VMOVSS (when the source operand is an XMM register and the destination is memory)
DEST[31:0] ← SRC[31:0]

MOVSS (Legacy SSE version when the source operand is memory and the destination is an XMM register)
DEST[31:0] ← SRC[31:0]
DEST[127:32] ← 0
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

MOVSS __m128 _mm_load_ss(float * p)
MOVSS void_mm_store_ss(float * p, __m128 a)
MOVSS __m128 _mm_move_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.vvvv != 1111B.

MOVUPD- Move Unaligned Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 10 /r MOVUPD xmm1, xmm2/m128	A	V/V	SSE2	Move unaligned packed double-precision floating-point from xmm2/mem to xmm1
66 0F 11 /r MOVUPD xmm2/m128, xmm1	B	V/V	SSE2	Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem
VEX.128.66.0F.WIG 10 /r VMOVUPD xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed double-precision floating-point from xmm2/mem to xmm1
VEX.128.66.0F.WIG 11 /r VMOVUPD xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem
VEX.256.66.0F.WIG 10 /r VMOVUPD ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed double-precision floating-point from ymm2/mem to ymm1
VEX.256.66.0F.WIG 11 /r VMOVUPD ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed double-precision floating-point from ymm1 to ymm2/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

VEX.256 encoded version:

Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

128-bit versions:

Moves 128 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VMOVUPD (VEX.256 encoded version)

DEST[255:0] ← SRC[255:0]

VMOVUPD (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]
DEST[255:128] ← 0

MOVUPD (128-bit load and register-copy form Legacy SSE version)

DEST[127:0] ← SRC[127:0]
DEST[255:128] (Unmodified)

(V)MOVUPD (128-bit store form)

DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVUPD __m256d _mm256_loadu_pd (__m256d * p);

VMOVUPD _mm256_storeu_pd(_m256d *p, __m256d a);

MOVUPD __m128d _mm_loadu_pd (__m128d * p);

MOVUPD _mm_storeu_pd(__m128d *p, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4
Note treatment of #AC varies; additionally

#UD	If VEX.vvvv != 1111B.

MOVUPS- Move Unaligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 10 /r MOVUPS xmm1, xmm2/m128	A	V/V	SSE	Move unaligned packed single-precision floating-point from xmm2/mem to xmm1
0F 11 /r MOVUPS xmm2/m128, xmm1	B	V/V	SSE	Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem
VEX.128.0F.WIG 10 /r VMOVUPS xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed single-precision floating-point from xmm2/mem to xmm1
VEX.128.0F.WIG 11 /r VMOVUPS xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem
VEX.256.0F.WIG 10 /r VMOVUPS ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed single-precision floating-point from ymm2/mem to ymm1
VEX.256.0F.WIG 11 /r VMOVUPS ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed single-precision floating-point from ymm1 to ymm2/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

VEX.256 encoded version:

Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

128-bit versions:

Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VMOVUPS (VEX.256 encoded version)

DEST[255:0] ← SRC[255:0]

VMOVUPS (VEX.128 encoded load-form)

DEST[127:0] ← SRC[127:0]
DEST[255:128] ← 0

MOVUPS (128-bit load and register-copy form Legacy SSE version)

DEST[127:0] ← SRC[127:0]
DEST[255:128] (Unmodified)

(V)MOVUPS (128-bit store form)

DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVUPS __m256 _mm256_loadu_ps (__m256 * p);

VMOVUPS _mm256_storeu_ps(_m256 *p, __m256 a);

MOVUPS __m128 _mm_loadu_ps (__m128 * p);

MOVUPS _mm_storeu_ps(__m128 *p, __m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4
Note treatment of #AC varies; additionally

#UD	If VEX.vvvv != 1111B.

MPSADBW - Multiple Sum of Absolute Differences

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 42 /r ib MPSADBW xmm1, xmm2/m128, imm8	A	V/V	SSE4_1	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in xmm1 and xmm2/m128 and writes the results in xmm1. Starting offsets within xmm1 and xmm2/m128 are determined by imm8
VEX.NDS.128.66.0F3A.WIG 42 /r ib VMPSADBW xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in xmm2 and xmm3/m128 and writes the results in xmm1. Starting offsets within xmm2 and xmm3/m128 are determined by imm8

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w))	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

MPSADBW sums the absolute difference of 4 unsigned bytes selected by immediate bits 0-1 from the second source with sequential groups of 4 unsigned bytes in the first source operand. The source bytes from the first source operand start at an offset determined by bit 2 of the immediate. The operation is repeated 8 times, each time using the same second source input but selecting the group of 4 bytes starting at the next higher byte in the first source. Each 16-bit sum is written to dest.

The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: The first source and destination are the same. Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

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

Operation

VMPSADBW (VEX.128 encoded version)

```
SRC2_OFFSET ← imm8[1:0]*32
SRC1_OFFSET ← imm8[2]*32
SRC1_BYTE0 ← SRC1[SRC1_OFFSET+7:SRC1_OFFSET]
SRC1_BYTE1 ← SRC1[SRC1_OFFSET+15:SRC1_OFFSET+8]
SRC1_BYTE2 ← SRC1[SRC1_OFFSET+23:SRC1_OFFSET+16]
SRC1_BYTE3 ← SRC1[SRC1_OFFSET+31:SRC1_OFFSET+24]
SRC1_BYTE4 ← SRC1[SRC1_OFFSET+39:SRC1_OFFSET+32]
SRC1_BYTE5 ← SRC1[SRC1_OFFSET+47:SRC1_OFFSET+40]
SRC1_BYTE6 ← SRC1[SRC1_OFFSET+55:SRC1_OFFSET+48]
SRC1_BYTE7 ← SRC1[SRC1_OFFSET+63:SRC1_OFFSET+56]
SRC1_BYTE8 ← SRC1[SRC1_OFFSET+71:SRC1_OFFSET+64]
SRC1_BYTE9 ← SRC1[SRC1_OFFSET+79:SRC1_OFFSET+72]
SRC1_BYTE10 ← SRC1[SRC1_OFFSET+87:SRC1_OFFSET+80]

SRC2_BYTE0 ←SRC2[SRC2_OFFSET+7:SRC2_OFFSET]
SRC2_BYTE1 ← SRC2[SRC2_OFFSET+15:SRC2_OFFSET+8]
SRC2_BYTE2 ← SRC2[SRC2_OFFSET+23:SRC2_OFFSET+16]
SRC2_BYTE3 ← SRC2[SRC2_OFFSET+31:SRC2_OFFSET+24]

TEMP0 ← ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[15:0] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[31:16] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[47:32] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[63:48] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE5 - SRC2_BYTE1)
```

TEMP2 ← ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[79:64] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE6 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[95:80] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[111:96] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[127:112] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

DEST[255:128] ← 0

MPSADBW (128-bit Legacy SSE version)
SRC_OFFSET ← imm8[1:0]*32
DEST_OFFSET ← imm8[2]*32
DEST_BYTE0 ← DEST[DEST_OFFSET+7:DEST_OFFSET]
DEST_BYTE1 ← DEST[DEST_OFFSET+15:DEST_OFFSET+8]
DEST_BYTE2 ← DEST[DEST_OFFSET+23:DEST_OFFSET+16]
DEST_BYTE3 ← DEST[DEST_OFFSET+31:DEST_OFFSET+24]
DEST_BYTE4 ← DEST[DEST_OFFSET+39:DEST_OFFSET+32]
DEST_BYTE5 ← DEST[DEST_OFFSET+47:DEST_OFFSET+40]
DEST_BYTE6 ← DEST[DEST_OFFSET+55:DEST_OFFSET+48]
DEST_BYTE7 ← DEST[DEST_OFFSET+63:DEST_OFFSET+56]
DEST_BYTE8 ← DEST[DEST_OFFSET+71:DEST_OFFSET+64]
DEST_BYTE9 ← DEST[DEST_OFFSET+79:DEST_OFFSET+72]
DEST_BYTE10 ← DEST[DEST_OFFSET+87:DEST_OFFSET+80]

SRC_BYTE0 ←SRC[SRC_OFFSET+7:SRC_OFFSET]
SRC_BYTE1 ← SRC[SRC_OFFSET+15:SRC_OFFSET+8]
SRC_BYTE2 ← SRC[SRC_OFFSET+23:SRC_OFFSET+16]
SRC_BYTE3 ← SRC[SRC_OFFSET+31:SRC_OFFSET+24]

TEMP0 ← ABS(DEST_BYTE0 - SRC_BYTE0)

```
TEMP1 ← ABS(DEST_BYTE1 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE2 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE3 - SRC_BYTE3)
DEST[15:0] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(DEST_BYTE1 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE2 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE3 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE4 - SRC_BYTE3)
DEST[31:16] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(DEST_BYTE2 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE3 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE4 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE5 - SRC_BYTE3)
DEST[47:32] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(DEST_BYTE3 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE4 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE5 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE6 - SRC_BYTE3)
DEST[63:48] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(DEST_BYTE4 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE5 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE6 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE7 - SRC_BYTE3)
DEST[79:64] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(DEST_BYTE5 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE6 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE7 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE8 - SRC_BYTE3)
DEST[95:80] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(DEST_BYTE6 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE7 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE8 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE9 - SRC_BYTE3)
DEST[111:96] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(DEST_BYTE7 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE8 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE9 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE10 - SRC_BYTE3)
DEST[127:112] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

MPSADBW __m128i _mm_mpsadbw_epu8 (__m128i s1, __m128i s2, const int mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1

MULPD- Multiply Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 59 /r MULPD xmm1, xmm2/m128	A	V/V	SSE2	Multiply packed double-precision floating-point values from xmm2/mem to xmm1 and stores result in xmm1
VEX.NDS.128.66.0F.WIG 59 /r VMULPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Multiply packed double-precision floating-point values from xmm3/mem to xmm2 and stores result in xmm1
VEX.NDS.256.66.0F.WIG 59 /r VMULPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Multiply packed double-precision floating-point values from ymm3/mem to ymm2 and stores result in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD Multiply of the two or four packed double-precision floating-point values from the first Source operand to the Second Source operand, and stores the packed double-precision floating-point results in the destination operand.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VMULPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
DEST[127:64] ← SRC1[127:64] * SRC2[127:64]
DEST[191:128] ← SRC1[191:128] * SRC2[191:128]
DEST[255:192] ← SRC1[255:192] * SRC2[255:192]

.

VMULPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
DEST[127:64] ← SRC1[127:64] * SRC2[127:64]
DEST[255:128] ← 0

MULPD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] * SRC[63:0]
DEST[127:64] ← DEST[127:64] * SRC[127:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMULPD __m256d _mm256_mul_pd (__m256d a, __m256d b);

MULPD __m128d _mm_mul_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

MULPS- Multiply Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 59 /r MULPS xmm1, xmm2/m128	A	V/V	SSE	Multiply packed single-precision floating-point values from xmm2/mem to xmm1 and stores result in xmm1
VEX.NDS.128.0F.WIG 59 /r VMULPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Multiply packed single-precision floating-point values from xmm3/mem to xmm2 and stores result in xmm1
VEX.NDS.256.0F.WIG 59 /r VMULPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Multiply packed single-precision floating-point values from ymm3/mem to ymm2 and stores result in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD multiply of the four or eight packed single-precision floating-point values from the first Source operand to the Second Source operand, and stores the packed double-precision floating-point results in the destination operand.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VMULPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[63:32] ← SRC1[63:32] * SRC2[63:32]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[127:96] ← SRC1[127:96] * SRC2[127:96]
DEST[159:128] ← SRC1[159:128] * SRC2[159:128]
DEST[191:160]← SRC1[191:160] * SRC2[191:160]
DEST[223:192] ← SRC1[223:192] * SRC2[223:192]
DEST[255:224] ← SRC1[255:224] * SRC2[255:224].

VMULPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[63:32] ← SRC1[63:32] * SRC2[63:32]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[127:96] ← SRC1[127:96] * SRC2[127:96]
DEST[255:128] ← 0

MULPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[63:32] ← SRC1[63:32] * SRC2[63:32]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[127:96] ← SRC1[127:96] * SRC2[127:96]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMULPS __m256 _mm256_mul_ps (__m256 a, __m256 b);

MULPS __m128 _mm_mul_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

MULSD- Multiply Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 59 /r MULSD xmm1,xmm2/m64	A	V/V	SSE2	Multiply the low double-precision floating-point value in xmm2/mem64 by low double precision floating-point value in xmm1.
VEX.NDS.LIG.F2.0F.WIG 59/r VMULSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Multiply the low double-precision floating-point value in xmm3/mem64 by low double precision floating-point value in xmm2.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the low double-precision floating-point value in the second source operand by the low double-precision floating-point value in the first source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source operand and the destination operands are XMM registers. The high quadword of the destination operand is copied from the high bits of the first source operand. See Figure 11-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a scalar double-precision floating-point operation.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VMULSD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] * SRC2[63:0]

DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0

MULSD (128-bit Legacy SSE version)
DEST[63:0] ← DEST[63:0] * SRC[63:0]
DEST[255:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

MULSD __m128d _mm_mul_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

MULSS- Multiply Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 59 /r MULSS xmm1,xmm2/m32	A	V/V	SSE	Multiply the low single-precision floating-point value in xmm2/mem by the low single-precision floating-point value in xmm1.
VEX.NDS.LIG.F3.0F.WIG 59 /r VMULSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Multiply the low single-precision floating-point value in xmm3/mem by the low single-precision floating-point value in xmm2.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the low single-precision floating-point value from the second source operand by the low single-precision floating-point value in the first source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source operand and the destination operands are XMM registers. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a scalar single-precision floating-point operation.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VMULSS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:32] ← SRC1[127:32]

DEST[255:128] ← 0

MULSS (128-bit Legacy SSE version)
DEST[31:0] ← DEST[31:0] * SRC[31:0]
DEST[255:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

MULSS __m128 _mm_mul_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Underflow, Overflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

ORPD- Bitwise Logical OR of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 56/r ORPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical OR of packed double-precision floating-point values in xmm1 and xmm2/mem
VEX.NDS.128.66.0F.WIG 56 /r VORPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in xmm2 and xmm3/mem
VEX.NDS.256.66.0F.WIG 56 /r VORPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical OR of the two or four packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

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

Operation

VORPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE OR SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE OR SRC2[255:192]

.

VORPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[255:128] ← 0

ORPD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] BITWISE OR SRC[63:0]
DEST[127:64] ← DEST[127:64] BITWISE OR SRC[127:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VORPD __m256d _mm256_or_pd (__m256d a, __m256d b);

ORPD __m128d _mm_or_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

ORPS- Bitwise Logical OR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 56 /r ORPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical OR of packed single-precision floating-point values in xmm1 and xmm2/mem
VEX.NDS.128.0F.WIG 56 /r VORPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in xmm2 and xmm3/mem
VEX.NDS.256.0F.WIG 56 /r VORPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical OR of the four or eight packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

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

Operation

VORPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[159:128] ← SRC1[159:128] BITWISE OR SRC2[159:128]
DEST[191:160]← SRC1[191:160] BITWISE OR SRC2[191:160]
DEST[223:192] ← SRC1[223:192] BITWISE OR SRC2[223:192]
DEST[255:224] ← SRC1[255:224] BITWISE OR SRC2[255:224].

VORPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[255:128] ← 0

ORPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VORPS __m256 _mm256_or_ps (__m256 a, __m256 b);

ORPS __m128 _mm_or_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PABSB/PABSW/PABSD - Packed Absolute Value

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 1C /r PABSB xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
66 0F 38 1D /r PABSW xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
66 0F 38 1E /r PABSD xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1C /r VPABSB xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1D /r VPABSW xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1E /r VPABSD xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

PABSB/W/D computes the absolute value of each data element of the source operand and stores the UNSIGNED results in the destination operand. PABSB operates on signed bytes, PABSW operates on signed 16-bit words, and PABSD operates on signed 32-bit integers. The source is an XMM register or a 128-bit memory location. The destination operand is an XMM register.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

```
BYTE_ABS(SRC)
{
    DEST [7:0]← ABS(SRC[7:0])
    .. repeat operation for 2nd through 15th bytes
    DEST [127..120]← ABS(SRC[127:120])
}

WORD_ABS(SRC)
{
    DEST [15:0]← ABS(SRC[15:0])
    .. repeat operation for 2nd through 7th 16-bit words
    DEST [127..112]← ABS(SRC[127:112])
}

DWORD_ABS(SRC)
{
    DEST [31:0]← ABS(SRC[31:0])
    DEST [63:32]← ABS(SRC[63:32])
    DEST [95:64]← ABS(SRC[95:64])
    DEST [127..96]← ABS(SRC[127:96])
}
```

VPABSB (VEX.128 encoded version)
DEST[127:0] ← BYTE_ABS(SRC)
DEST[255:128] ← 0

PABSB (128-bit Legacy SSE version)
DEST[127:0] ← BYTE_ABS(SRC)
DEST[255:128] (Unmodified)

VPABSW (VEX.128 encoded version)
DEST[127:0] ← WORD_ABS(SRC)
DEST[255:128] ← 0

PABSW (128-bit Legacy SSE version)
DEST[127:0] ← WORD_ABS(SRC)
DEST[255:128] (Unmodified)

VPABSD (VEX.128 encoded version)
DEST[127:0] ← DWORD_ABS(SRC)
DEST[255:128] ← 0

PABSD (128-bit Legacy SSE version)
DEST[127:0] ← DWORD_ABS(SRC)
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PABSB __m128i _mm_abs_epi8 (__m128i a)

PABSW __m128i _mm_abs_epi16 (__m128i a)

PABSD __m128i _mm_abs_epi32 (__m128i a)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

PACKSSWB/PACKSSDW- Pack with Signed Saturation

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 63 /r PACKSSWB xmm1,xmm2/m128	A	V/V	SSE2	Converts 8 packed signed word integers from xmm1 and from xmm2/m128 into 16 packed signed byte integers in xmm1 using signed saturation.
66 0F 6B /r PACKSSDW xmm1,xmm2/m128	A	V/V	SSE2	Converts 4 packed signed doubleword integers from xmm1 and from xmm2/m128 into 8 packed signed word integers in xmm1 using signed saturation.
VEX.NDS.128.66.0F.WIG 63 /r VPACKSSWB xmm1,xmm2, xmm3/m128	B	V/V	AVX	Converts 8 packed signed word integers from xmm2 and from xmm3/m128 into 16 packed signed byte integers in xmm1 using signed saturation.
VEX.NDS.128.66.0F.WIG 6B /r VPACKSSDW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Converts 4 packed signed doubleword integers from xmm2 and from xmm3/m128 into 8 packed signed word integers in xmm1 using signed saturation.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 5-17 for an example of the packing operation.

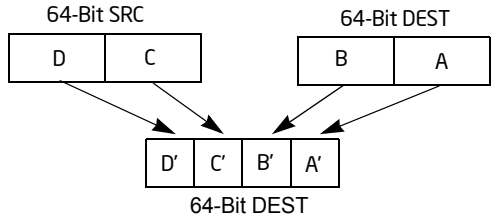


Figure 5-17. PACKSSDW Instruction Operation using 64-bit Operands

The PACKSSWB instruction converts 8 signed word integers from the first source operand and 8 signed word integers from the second source operand into 16 signed byte integers and stores the result in the destination operand. If a signed word integer value is beyond the range of a signed byte integer (that is, greater than 7FH for a positive integer or greater than 80H for a negative integer), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination.

The PACKSSDW instruction packs 4 signed doublewords from the first source operand and 4 signed doublewords from the second source operand into 8 signed words in the destination operand (see Figure 5-17).

If a signed doubleword integer value is beyond the range of a signed word (that is, greater than 7FFFH for a positive integer or greater than 8000H for a negative integer), the saturated signed word integer value of 7FFFH or 8000H, respectively, is stored into the destination.

When operating on 128-bit operands, the first source and destination operands are XMM registers. and the second source operand can be either an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
SATURATING_PACK_WB(SRC1, SRC2)
DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0])
DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16])
DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32])
DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48])
DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64])
DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80])
DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96])
DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112])
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0])
```

DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16])
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32])
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48])
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64])
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80])
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96])
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112])

SATURATING_PACK_DW(SRC1, SRC2)
DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0])
DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32])
DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64])
DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96])
DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0])
DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32])
DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64])
DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96])

PACKSSDW
DEST[127:0] ← SATURATING_PACK_DW(DEST, SRC)
DEST[255:128] (Unmodified)

VPACKSSDW
DEST[127:0] ← SATURATING_PACK_DW(SRC1, SRC2)
DEST[255:128] ← 0

PACKSSWB
DEST[127:0] ← SATURATING_PACK_WB(DEST, SRC)
DEST[255:128] (Unmodified)

VPACKSSWB
DEST[127:0] ← SATURATING_PACK_WB(SRC1, SRC2)
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PACKSSWB __m128i _mm_packs_epi16(__m128i m1, __m128i m2)

PACKSSDW __m128i _mm_packs_epi32(__m128i m1, __m128i m2)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PACKUSWB/PACKUSDW- Pack with Unsigned Saturation

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 67 /r PACKUSWB xmm1,xmm2/m128	A	V/V	SSE2	Converts 8 signed word integers from xmm1 and 8 signed word integers from xmm2/m128 into 16 unsigned byte integers in xmm1 using unsigned saturation.
66 0F 38 2B /r PACKUSDW xmm1, xmm2/m128	A	V/V	SSE4_1	Convert 4 packed signed doubleword integers from xmm1 and 4 packed signed doubleword integers from xmm2/m128 into 8 packed unsigned word integers in xmm1 using unsigned saturation.
VEX.NDS.128.66.0F.WIG 67 /r VPACKUSWB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Converts 8 signed word integers from xmm2 and 8 signed word integers from xmm3/m128 into 16 unsigned byte integers in xmm1 using unsigned saturation.
VEX.NDS.128.66.0F38.WIG 2B /r VPACKUSDW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Convert 4 packed signed doubleword integers from xmm2 and 4 packed signed doubleword integers from xmm3/m128 into 8 packed unsigned word integers in xmm1 using unsigned saturation.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

packuswb:

Converts 8 signed word integers from the second source operand and 8 signed word integers from the first source operand into 8 unsigned byte integers and stores the result in the destination operand. (See Figure 5-17 for an example of the packing

operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

The first source operand and destination operand must be an XMM register and the second source operand can be either an XMM register or a 128-bit memory location.

packusdw:

Converts packed signed doubleword integers into packed unsigned word integers using unsigned saturation to handle overflow conditions. If the signed doubleword value is beyond the range of an unsigned word (that is, greater than FFFFH or less than 0000H), the saturated unsigned word integer value of FFFFH or 0000H, respectively stored in the destination.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
StaurateSignedWordToUnsignedByte(SRC)
{
    TMP ← SRC < 0 ? 0 : SRC
    return SRC > FFH ? FFH: TMP
}

SaturateSignedDWordToUnsignedWord(SRC)
{
    TMP ← SRC < 0 ? 0 : SRC
    return SRC > FFFFH ? FFFFH: TMP
}

UNSIGNED_SATURATING_PACK_DW(SRC1, SRC2)
DEST[15:0] ← SaturateSignedDWordToUnsignedWord(SRC1[31:0])
DEST[31:16] ← SaturateSignedDWordToUnsignedWord(SRC1[63:32])
DEST[47:32] ← SaturateSignedDWordToUnsignedWord(SRC1[95:64])
DEST[63:48] ← SaturateSignedDWordToUnsignedWord(SRC1[127:96])
DEST[79:64] ← SaturateSignedDWordToUnsignedWord(SRC2[31:0])
DEST[95:80] ← SaturateSignedDWordToUnsignedWord(SRC2[63:32])
DEST[111:96] ← SaturateSignedDWordToUnsignedWord(SRC2[95:64])
DEST[127:112] ← SaturateSignedDWordToUnsignedWord(SRC2[127:96])

UNSIGNED_SATURATING_PACK_WB(SRC1, SRC2)
DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0])
DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16])
```

DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32])
DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48])
DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64])
DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80])
DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96])
DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112])
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0])
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16])
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32])
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48])
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64])
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80])
DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96])
DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112])

VPACKUSWB (VEX.128 encoded version)
DEST[127:0] ← UNSIGNED_SATURATING_PACK_WB(SRC1, SRC2)
DEST[255:128] ← 0

VPACKUSDW (VEX.128 encoded version)
DEST[127:0] ← UNSIGNED_SATURATING_PACK_DW(SRC1, SRC2)
DEST[255:128] ← 0

PACKUSWB (128-bit Legacy SSE version)
DEST[127:0] ← UNSIGNED_SATURATING_PACK_WB(DEST, SRC)
DEST[255:128] (Unmodified)

PACKUSDW (128-bit Legacy SSE version)
DEST[127:0] ← UNSIGNED_SATURATING_PACK_DW(DEST, SRC)
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PACKUSDW __m128i _mm_packus_epi32(__m128i m1, __m128i m2);

PACKUSWB __m128i _mm_packus_epi16(__m128i m1, __m128i m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PADDB/PADDW/PADDD/PADDQ- Add Packed Integers

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F FC /r PADDB xmm1, xmm2/m128	A	V/V	SSE2	Add packed byte integers from xmm2/m128 and xmm1.
66 0F FD /r PADDW xmm1, xmm2/m128	A	V/V	SSE2	Add packed word integers from xmm2/m128 and xmm1.
66 0F FE /r PADDD xmm1, xmm2/m128	A	V/V	SSE2	Add packed doubleword integers from xmm2/m128 and xmm1.
66 0F D4 /r PADDQ xmm1, xmm2/m128	A	V/V	SSE2	Add packed quadword integers xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG FC /r VPADDB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed byte integers from xmm3/m128 and xmm2.
VEX.NDS.128.66.0F.WIG FD /r VPADDW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed word integers from xmm3/m128 and xmm2.
VEX.NDS.128.66.0F.WIG FE /r VPADDD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed doubleword integers from xmm3/m128 and xmm2.
VEX.NDS.128.66.0F.WIG D4 /r VPADDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed quadword integers xmm3/m128 and xmm2.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Adds the packed byte, word, doubleword, or quadword integers in the first source operand to the second source operand and stores the result in the destination operand. The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operand are XMM registers. When a result is too large to be represented in the 8/16/32/64 integer (overflow), the result is wrapped around and the low bits are written to the destination element (that is, the carry is ignored).

Note that these instructions can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPADDB (VEX.128 encoded version)

DEST[7:0] ← SRC1[7:0]+SRC2[7:0]
DEST[15:8] ← SRC1[15:8]+SRC2[15:8]
DEST[23:16] ← SRC1[23:16]+SRC2[23:16]
DEST[31:24] ← SRC1[31:24]+SRC2[31:24]
DEST[39:32] ← SRC1[39:32]+SRC2[39:32]
DEST[47:40] ← SRC1[47:40]+SRC2[47:40]
DEST[55:48] ← SRC1[55:48]+SRC2[55:48]
DEST[63:56] ← SRC1[63:56]+SRC2[63:56]
DEST[71:64] ← SRC1[71:64]+SRC2[71:64]
DEST[79:72] ← SRC1[79:72]+SRC2[79:72]
DEST[87:80] ← SRC1[87:80]+SRC2[87:80]
DEST[95:88] ← SRC1[95:88]+SRC2[95:88]
DEST[103:96] ← SRC1[103:96]+SRC2[103:96]
DEST[111:104] ← SRC1[111:104]+SRC2[111:104]
DEST[119:112] ← SRC1[119:112]+SRC2[119:112]
DEST[127:120] ← SRC1[127:120]+SRC2[127:120]
DEST[255:128] ← 0

PADDB (128-bit Legacy SSE version)
DEST[7:0] ← DEST[7:0]+SRC[7:0]
DEST[15:8] ← DEST[15:8]+SRC[15:8]
DEST[23:16] ← DEST[23:16]+SRC[23:16]
DEST[31:24] ← DEST[31:24]+SRC[31:24]
DEST[39:32] ← DEST[39:32]+SRC[39:32]
DEST[47:40] ← DEST[47:40]+SRC[47:40]
DEST[55:48] ← DEST[55:48]+SRC[55:48]
DEST[63:56] ← DEST[63:56]+SRC[63:56]
DEST[71:64] ← DEST[71:64]+SRC[71:64]
DEST[79:72] ← DEST[79:72]+SRC[79:72]
DEST[87:80] ← DEST[87:80]+SRC[87:80]
DEST[95:88] ← DEST[95:88]+SRC[95:88]
DEST[103:96] ← DEST[103:96]+SRC[103:96]
DEST[111:104] ← DEST[111:104]+SRC[111:104]
DEST[119:112] ← DEST[119:112]+SRC[119:112]
DEST[127:120] ← DEST[127:120]+SRC[127:120]
DEST[255:128] (Unmodified)

VPADDW (VEX.128 encoded version)
DEST[15:0] ← SRC1[15:0]+SRC2[15:0]
DEST[31:16] ← SRC1[31:16]+SRC2[31:16]
DEST[47:32] ← SRC1[47:32]+SRC2[47:32]
DEST[63:48] ← SRC1[63:48]+SRC2[63:48]
DEST[79:64] ← SRC1[79:64]+SRC2[79:64]
DEST[95:80] ← SRC1[95:80]+SRC2[95:80]
DEST[111:96] ← SRC1[111:96]+SRC2[111:96]
DEST[127:112] ← SRC1[127:112]+SRC2[127:112]
DEST[255:128] ← 0

PADDW (128-bit Legacy SSE version)
DEST[15:0] ← DEST[15:0]+SRC[15:0]
DEST[31:16] ← DEST[31:16]+SRC[31:16]
DEST[47:32] ← DEST[47:32]+SRC[47:32]
DEST[63:48] ← DEST[63:48]+SRC[63:48]
DEST[79:64] ← DEST[79:64]+SRC[79:64]
DEST[95:80] ← DEST[95:80]+SRC[95:80]
DEST[111:96] ← DEST[111:96]+SRC[111:96]
DEST[127:112] ← DEST[127:112]+SRC[127:112]
DEST[255:128] (Unmodified)

VPADDD (VEX.128 encoded version)
DEST[31:0] ← SRC1[31:0]+SRC2[31:0]

DEST[63:32] ← SRC1[63:32]+SRC2[63:32]
DEST[95:64] ← SRC1[95:64]+SRC2[95:64]
DEST[127:96] ← SRC1[127:96]+SRC2[127:96]
DEST[255:128] ← 0

PADDD (128-bit Legacy SSE version)
DEST[31:0] ← DEST[31:0]+SRC[31:0]
DEST[63:32] ← DEST[63:32]+SRC[63:32]
DEST[95:64] ← DEST[95:64]+SRC[95:64]
DEST[127:96] ← DEST[127:96]+SRC[127:96]
DEST[255:128] (Unmodified)

VPADDQ (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0]+SRC2[63:0]
DEST[127:64] ← SRC1[127:64]+SRC2[127:64]
DEST[255:128] ← 0

PADDQ (128-bit Legacy SSE version)
DEST[63:0] ← DEST[63:0]+SRC[63:0]
DEST[127:64] ← DEST[127:64]+SRC[127:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PADDB __m128i_mm_add_epi8 (__m128ia,__m128ib)

PADDW __m128i _mm_add_epi16 (__m128i a, __m128i b)

PADDD __m128i _mm_add_epi32 (__m128i a, __m128i b)

PADDQ __m128i _mm_add_epi64 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PADDSB/PADDSW- Add Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F EC /r PADDSB xmm1, xmm2/m128	A	V/V	SSE2	Add packed signed byte integers from xmm2/m128 and xmm1 saturate the results.
66 0F ED /r PADDSW xmm1, xmm2/m128	A	V/V	SSE2	Add packed signed word integers from xmm2/m128 and xmm1 and saturate the results.
VEX.NDS.128.66.0F.WIG EC /r VPADDSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed signed byte integers from xmm3/m128 and xmm2 saturate the results.
VEX.NDS.128.66.0F.WIG ED /r VPADDSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed signed word integers from xmm3/m128 and xmm2 and saturate the results.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD add of the packed signed integers from the second source operand and the first source operand and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation.

Overflow is handled with signed saturation, as described in the following paragraphs. The second source operand can be either an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers.

The PADDSB instruction adds packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The PADDSW instruction adds packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPADDSB

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[255:128] ← 0
```

PADDSB

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[111:120] + SRC[127:120]);
DEST[255:128] (Unmodified)
```

VPADDSW

```
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] + SRC2[127:112]);
DEST[255:128] ← 0
```

PADDSW

```
DEST[15:0] ← SaturateToSignedWord (DEST[15:0] + SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (DEST[127:112] + SRC[127:112]);
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

PADDSB __m128i _mm_adds_epi8 (__m128i a, __m128i b)

PADDSW __m128i _mm_adds_epi16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PADDUSB/PADDUSW- Add Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F DC /r PADDUSB xmm1, xmm2/m128	A	V/V	SSE2	Add packed unsigned byte integers from xmm2/m128 and xmm1 saturate the results.
66 0F DD /r PADDUSW xmm1, xmm2/m128	A	V/V	SSE2	Add packed unsigned word integers from xmm2/m128 to xmm1 and saturate the results.
VEX.NDS.128.660F.WIG DC /r VPADDUSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed unsigned byte integers from xmm3/m128 to xmm2 and saturate the results.
VEX.NDS.128.66.0F.WIG DD /r VPADDUSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed unsigned word integers from xmm3/m128 to xmm2 and saturate the results.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD add of the packed unsigned integers from the second source operand and the first source operand and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

The first source operand and the destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location.

The PADDUSB instruction adds packed unsigned byte integers. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand. The PADDUSW instruction adds packed unsigned word integers. When an individual word result is

beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPADDUSB

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[255:128] ← 0
```

PADDUSB

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (DEST[111:120] + SRC[127:120]);
DEST[255:128] (Unmodified)
```

VPADDUSW

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] + SRC2[127:112]);
DEST[255:128] ← 0
```

PADDUSW

```
DEST[15:0] ← SaturateToSUnsgnedWord (DEST[15:0] + SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (DEST[127:112] + SRC[127:112]);
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

PADDUSB __m128i _mm_adds_epu8 (__m128i a, __m128i b)

PADDUSW __m128i _mm_adds_epu16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PALIGNR - Byte Align

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0F /r ib PALIGNR xmm1, xmm2/m128, imm8	A	V/V	SSSE3	Concatenate destination and source operands, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1
VEX.NDS.128.66.0F3A.WIG 0F /r ib VPALIGNR xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Concatenate xmm2 and xmm3/m128, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PALIGNR concatenates the first source operand and the second source operand into an intermediate composite, shifts the composite at byte granularity to the right by a constant immediate, and extracts the right aligned result into the destination. The first source and destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. The immediate value is considered unsigned. Immediate shift counts larger than 32 for 128-bit operands produces a zero result.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PALIGNR

temp1[255:0] ← CONCATENATE(DEST,SRC)>>(imm8*8)
DEST[127:0] ← temp1[127:0]
DEST[255:128] (Unmodified)

VPALIGNR
temp1[255:0] ← CONCATENATE(SRC1,SRC2)>>(imm8*8)
DEST[127:0] ← temp1[127:0]
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PALIGNR __m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PAND- Logical AND

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F DB /r PAND xmm1, xmm2/m128	A	V/V	SSE2	Bitwise AND of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG DB /r VPAND xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise AND of xmm3/m128 and xmm2.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND operation on the second source operand and the first source operand and stores the result in the destination operand. The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPAND (VEX.128 encoded version)

DEST ← SRC1 AND SRC2
DEST[255:128] ← 0

PAND (128-bit Legacy SSE version)

DEST ← DEST AND SRC
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PAND __m128i _mm_and_si128 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PANDN- Logical AND NOT

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F DF /r PANDN xmm1, xmm2/m128	A	V/V	SSE2	Bitwise AND NOT of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG DF /r VPANDN xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise AND NOT of xmm3/m128 and xmm2.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical NOT operation on the first source operand and then performs a bitwise logical AND with the second source operand and stores the result in the destination operand. The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPANDN (VEX.128 encoded version)

DEST ← NOT(SRC1) AND SRC2
DEST[255:128] ← 0

PANDN(128-bit Legacy SSE version)

DEST ← NOT(DEST) AND SRC
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PANDN __m128i _mm_andnot_si128 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PAVGB/PAVGW - Average Packed Integers

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E0 /r PAVGB xmm1, xmm2/m128	A	V/V	SSE2	Average packed unsigned byte integers from xmm2/m128 and xmm1 with rounding.
66 0F E3 /r PAVGW xmm1, xmm2/m128	A	V/V	SSE2	Average packed unsigned word integers from xmm2/m128 and xmm1 with rounding.
VEX.NDS.128.66.0F.WIG E0 /r VPAVGB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Average packed unsigned byte integers from xmm3/m128 and xmm2 with rounding.
VEX.NDS.128.66.0F.WIG E3 /r VPAVGW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Average packed unsigned word integers from xmm3/m128 and xmm2 with rounding.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD average of the packed unsigned integers from the second source operand and the first source operand and stores the results in the destination operand. For each corresponding pair of data elements in the first and second source operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position. The destination and first source operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location.

The PAVGB instruction operates on packed unsigned bytes and the PAVGW instruction operates on packed unsigned words.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPAVGB (VEX.128 encoded version)

```
DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1;
(* Repeat operation performed for bytes 2 through 15 *)
DEST[127:120] ← (SRC1[127:120] + SRC2[127:120] + 1) >> 1
DEST[255:128] ← 0
```

PAVGB (128-bit Legacy SSE version)

```
DEST[7:0] ← (SRC[7:0] + DEST[7:0] + 1) >> 1;
(* Repeat operation performed for bytes 2 through 15 *)
DEST[127:120] ← (SRC[127:120] + DEST[127:120] + 1) >> 1
DEST[255:128] (Unmodified)
```

VPAVGW (VEX.128 encoded version)

```
DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1;
(* Repeat operation performed for 16-bit words 2 through 7 *)
DEST[127:112] ← (SRC1[127:112] + SRC2[127:112] + 1) >> 1
DEST[255:128] ← 0
```

PAVGW (128-bit Legacy SSE version)

```
DEST[15:0] ← (SRC[15:0] + DEST[15:0] + 1) >> 1;
(* Repeat operation performed for 16-bit words 2 through 7 *)
DEST[127:112 ← (SRC[127:112] + DEST[127:112] + 1) >> 1
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

PAVGB __m128i _mm_avg_epu8 (__m128i a, __m128i b)

PAVGW __m128i _mm_avg_epu16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PBLENDVB - Variable Blend Packed Bytes

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 10 /r PBLENDVB xmm1, xmm2/m128, <XMM0>	A	V/V	SSE4_1	Select byte values from xmm1 and xmm2/m128 using mask bits in the implicit mask register, XMM0, and store the values into xmm1
VEX.NDS.128.66.0F3A.W0 4C /r /is4 VPBLENDVB xmm1, xmm2, xmm3/m128, xmm4	B	V/V	AVX	Select byte values from xmm2 and xmm3/m128 using mask bits in the specified mask register, xmm4, and store the values into xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

Description

Conditionally copy byte elements from the second source operand and the first source operand depending on mask bits defined in the mask register operand. The mask bits are the most significant bit in each byte element of the mask register.

Each byte element of the destination operand is copied from:

- the corresponding byte element in the second source operand, If a mask bit is "1"; or
- the corresponding byte element in the first source operand, If a mask bit is "0"

The register assignment of the implicit third operand is defined to be the architectural register XMM0

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (255:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute PBLENDVB with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (255:128) of the corresponding YMM register

(destination register) are zeroed. VEX.L must be 0, otherwise the instruction will #UD. VEX.W must be 0, otherwise, the instruction will #UD.

VPBLENDVB permits the mask to be any XMM or YMM register. In contrast, PBLENDVB treats XMM0 implicitly as the mask and do not support non-destructive destination operation. An attempt to execute PBLENDVB encoded with a VEX prefix will cause a #UD exception.

Operation

VPBLENDVB (VEX.128 encoded version)

```
MASK ← SRC3
IF (MASK[7] == 1) THEN DEST[7:0] ← SRC2[7:0];
ELSE DEST[7:0] ← SRC1[7:0];
IF (MASK[15] == 1) THEN DEST[15:8] ← SRC2[15:8];
ELSE DEST[15:8] ← SRC1[15:8];
IF (MASK[23] == 1) THEN DEST[23:16] ← SRC2[23:16]
ELSE DEST[23:16] ← SRC1[23:16];
IF (MASK[31] == 1) THEN DEST[31:24] ← SRC2[31:24]
ELSE DEST[31:24] ← SRC1[31:24];
IF (MASK[39] == 1) THEN DEST[39:32] ← SRC2[39:32]
ELSE DEST[39:32] ← SRC1[39:32];
IF (MASK[47] == 1) THEN DEST[47:40] ← SRC2[47:40]
ELSE DEST[47:40] ← SRC1[47:40];
IF (MASK[55] == 1) THEN DEST[55:48] ← SRC2[55:48]
ELSE DEST[55:48] ← SRC1[55:48];
IF (MASK[63] == 1) THEN DEST[63:56] ← SRC2[63:56]
ELSE DEST[63:56] ← SRC1[63:56];
IF (MASK[71] == 1) THEN DEST[71:64] ← SRC2[71:64]
ELSE DEST[71:64] ← SRC1[71:64];
IF (MASK[79] == 1) THEN DEST[79:72] ← SRC2[79:72]
ELSE DEST[79:72] ← SRC1[79:72];
IF (MASK[87] == 1) THEN DEST[87:80] ← SRC2[87:80]
ELSE DEST[87:80] ← SRC1[87:80];
IF (MASK[95] == 1) THEN DEST[95:88] ← SRC2[95:88]
ELSE DEST[95:88] ←    SRC1[95:88];
IF (MASK[103] == 1) THEN DEST[103:96] ← SRC2[103:96]
ELSE DEST[103:96] ←    SRC1[103:96];
IF (MASK[111] == 1) THEN DEST[111:104] ← SRC2[111:104]
ELSE DEST[111:104] ← SRC1[111:104];
IF (MASK[119] == 1) THEN DEST[119:112] ← SRC2[119:112]
ELSE DEST[119:112] ← SRC1[119:112];
IF (MASK[127] == 1) THEN DEST[127:120] ← SRC2[127:120]
ELSE DEST[127:120] ← SRC1[127:120])
DEST[255:128] ← 0
```

PBLENDVB (128-bit Legacy SSE version)

```
MASK ← XMM0
IF (MASK[7] == 1) THEN DEST[7:0] ← SRC[7:0];
ELSE DEST[7:0] ← DEST[7:0];
IF (MASK[15] == 1) THEN DEST[15:8] ← SRC[15:8];
ELSE DEST[15:8] ← DEST[15:8];
IF (MASK[23] == 1) THEN DEST[23:16] ← SRC[23:16]
ELSE DEST[23:16] ← DEST[23:16];
IF (MASK[31] == 1) THEN DEST[31:24] ← SRC[31:24]
ELSE DEST[31:24] ← DEST[31:24];
IF (MASK[39] == 1) THEN DEST[39:32] ← SRC[39:32]
ELSE DEST[39:32] ← DEST[39:32];
IF (MASK[47] == 1) THEN DEST[47:40] ← SRC[47:40]
ELSE DEST[47:40] ← DEST[47:40];
IF (MASK[55] == 1) THEN DEST[55:48] ← SRC[55:48]
ELSE DEST[55:48] ← DEST[55:48];
IF (MASK[63] == 1) THEN DEST[63:56] ← SRC[63:56]
ELSE DEST[63:56] ← DEST[63:56];
IF (MASK[71] == 1) THEN DEST[71:64] ← SRC[71:64]
ELSE DEST[71:64] ← DEST[71:64];
IF (MASK[79] == 1) THEN DEST[79:72] ← SRC[79:72]
ELSE DEST[79:72] ← DEST[79:72];
IF (MASK[87] == 1) THEN DEST[87:80] ← SRC[87:80]
ELSE DEST[87:80] ← DEST[87:80];
IF (MASK[95] == 1) THEN DEST[95:88] ← SRC[95:88]
ELSE DEST[95:88] ←    DEST[95:88];
IF (MASK[103] == 1) THEN DEST[103:96] ← SRC[103:96]
ELSE DEST[103:96] ←    DEST[103:96];
IF (MASK[111] == 1) THEN DEST[111:104] ← SRC[111:104]
ELSE DEST[111:104] ← DEST[111:104];
IF (MASK[119] == 1) THEN DEST[119:112] ← SRC[119:112]
ELSE DEST[119:112] ← DEST[119:112];
IF (MASK[127] == 1) THEN DEST[127:120] ← SRC[127:120]
ELSE DEST[127:120] ← DEST[127:120])
```

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PBLENDVB __m128i _mm_blendv_epi8 (__m128i v1, __m128i v2, __m128i mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.W = 1.

PBLENDW - Blend Packed Words

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0E /r ib PBLENDW xmm1, xmm2/m128, imm8	A	V/V	SSE4_1	Select words from xmm1 and xmm2/m128 from mask specified in imm8 and store the values into xmm1
VEX.NDS.128.66.0F3A.WIG 0E /r ib VPBLENDW xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Select words from xmm2 and xmm3/m128 from mask specified in imm8 and store the values into xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Words from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word is unchanged.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPBLENDW (VEX.128 encoded version)

IF (imm8[0] == 1) THEN DEST[15:0] ← SRC2[15:0]
ELSE DEST[15:0] ← SRC1[15:0]
IF (imm8[1] == 1) THEN DEST[31:16] ← SRC2[31:16]
ELSE DEST[31:16] ← SRC1[31:16]

IF (imm8[2] == 1) THEN DEST[47:32] ← SRC2[47:32]
ELSE DEST[47:32] ← SRC1[47:32]
IF (imm8[3] == 1) THEN DEST[63:48] ← SRC2[63:48]
ELSE DEST[63:48] ← SRC1[63:48]
IF (imm8[4] == 1) THEN DEST[79:64] ← SRC2[79:64]
ELSE DEST[79:64] ← SRC1[79:64]
IF (imm8[5] == 1) THEN DEST[95:80] ← SRC2[95:80]
ELSE DEST[95:80] ← SRC1[95:80]
IF (imm8[6] == 1) THEN DEST[111:96] ← SRC2[111:96]
ELSE DEST[111:96] ← SRC1[111:96]
IF (imm8[7] == 1) THEN DEST[127:112] ← SRC2[127:112]
ELSE DEST[127:112] ← SRC1[127:112]
DEST[255:128] ← 0

PBLENDW (128-bit Legacy SSE version)
IF (imm8[0] == 1) THEN DEST[15:0] ← SRC[15:0]
ELSE DEST[15:0] ← DEST[15:0]
IF (imm8[1] == 1) THEN DEST[31:16] ← SRC[31:16]
ELSE DEST[31:16] ← DEST[31:16]
IF (imm8[2] == 1) THEN DEST[47:32] ← SRC[47:32]
ELSE DEST[47:32] ← DEST[47:32]
IF (imm8[3] == 1) THEN DEST[63:48] ← SRC[63:48]
ELSE DEST[63:48] ← DEST[63:48]
IF (imm8[4] == 1) THEN DEST[79:64] ← SRC[79:64]
ELSE DEST[79:64] ← DEST[79:64]
IF (imm8[5] == 1) THEN DEST[95:80] ← SRC[95:80]
ELSE DEST[95:80] ← DEST[95:80]
IF (imm8[6] == 1) THEN DEST[111:96] ← SRC[111:96]
ELSE DEST[111:96] ← DEST[111:96]
IF (imm8[7] == 1) THEN DEST[127:112] ← SRC[127:112]
ELSE DEST[127:112] ← DEST[127:112]

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PBLENDW __m128i _mm_blend_epi16 (__m128i v1, __m128i v2, const int mask)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PCLMULQDQ - Carry-Less Multiplication Quadword

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 44 /r ib PCLMULQDQ xmm1, xmm2/m128, imm8	A	V/V	CLMUL	Carry-less multiplication of one quadword of xmm1 by one quadword of xmm2/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm1 and xmm2/m128 should be used
VEX.NDS.128.66.0F3A.WIG 44 /r ib VPCLMULQDQ xmm1, xmm2, xmm3/m128, imm8	B	V/V	Both CLMUL and AVX flags	Carry-less multiplication of one quadword of xmm2 by one quadword of xmm3/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm2 and xmm3/m128 should be used

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a carry-less multiplication of two quadwords, selected from the first source and second source operand according to the value of the immediate byte. Bits 4 and 0 are used to select which 64-bit half of each operand to use according to Table 5-18, other bits of the immediate byte are ignored.

Table 5-18. PCLMULQDQ Quadword Selection of Immediate Byte

Imm[4]	Imm[0]	PCLMULQDQ Operation
0	0	CL_MUL(SRC2[1][63:0], SRC1[63:0])
0	1	CL_MUL(SRC2[63:0], SRC1[127:64])
1	0	CL_MUL(SRC2[127:64], SRC1[63:0])
1	1	CL_MUL(SRC2[127:64], SRC1[127:64])

NOTES:

1. SRC2 denotes the second source operand, which can be a register or memory; SRC1 denotes the first source and destination operand.

The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (255:128) of the corresponding YMM destination register remain unchanged.

Compilers and assemblers may implement the following pseudo-op syntax to simply programming and emit the required encoding for Imm8.

Table 5-19. Pseudo-Op and PCLMULQDQ Implementation

Pseudo-Op	Pseudo-Op	Imm8 Encoding
PCLMULLQLQDQ *xmm1, xmm2*	**VPCLMULLQLQDQ *xmm1, xmm2, xmm3***	**0000_0000B**
PCLMULHQLQDQ *xmm1, xmm2*	**VPCLMULHQLQDQ *xmm1, xmm2, xmm3***	**0000_0001B**
PCLMULLQHDQ *xmm1, xmm2*	**VPCLMULLQHDQ *xmm1, xmm2, xmm3***	**0001_0000B**
PCLMULHQHDQ *xmm1, xmm2*	**VPCLMULHQHDQ *xmm1, xmm2, xmm3***	**0001_0001B**

Operation

VPCLMULQDQ

```
IF (Imm8[0] = 0 )
    THEN
        TEMP1 ← SRC1 [63:0];
    ELSE
        TEMP1 ← SRC1 [127:64];
FI
IF (Imm8[4] = 0 )
    THEN
        TEMP2 ← SRC2 [63:0];
    ELSE
        TEMP2 ← SRC2 [127:64];
FI
For i = 0 to 63 {
    TmpB [ i ] ← (TEMP1[ 0 ] and TEMP2[ i ]);
    For j = 1 to i {
        TmpB [i] ← TmpB [i] xor (TEMP1[ j ] and TEMP2[ i - j ])
```

```
    }
    DEST[i] ← TmpB[i];
}
For i = 64 to 126 {
    TmpB [ i ] ← 0;
    For j = i - 63 to 63 {
        TmpB [i] ← TmpB [i] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[i] ← TmpB[i];
}
DEST[255:127] ← 0;
```

PCLMULQDQ

```
IF (Imm8[0] = 0 )
    THEN
        TEMP1 ← SRC1 [63:0];
    ELSE
        TEMP1 ← SRC1 [127:64];
FI
IF (Imm8[4] = 0 )
    THEN
        TEMP2 ← SRC2 [63:0];
    ELSE
        TEMP2 ← SRC2 [127:64];
FI
For i = 0 to 63 {
    TmpB [ i ] ← (TEMP1[ 0 ] and TEMP2[ i ]);
    For j = 1 to i {
        TmpB [i] ← TmpB [i] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[i] ← TmpB[i];
}
For i = 64 to 126 {
    TmpB [ i ] ← 0;
    For j = i - 63 to 63 {
        TmpB [i] ← TmpB [i] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[i] ← TmpB[i];
}
DEST[127] ← 0;
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PCLMULQDQ __m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PCMPEQB/PCMPEQW/PCMPEQD/PCMPEQQ- Compare Packed Integers for Equality

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 74 /r PCMPEQB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed bytes in xmm2/m128 and xmm1 for equality.
66 0F 75 /r PCMPEQW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed words in xmm2/m128 and xmm1 for equality.
66 0F 76 /r PCMPEQD xmm1, xmm2/m128	A	V/V	SSE2	Compare packed doublewords in xmm2/m128 and xmm1 for **equality.**
66 0F 38 29 /r PCMPEQQ xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed quadwords in xmm2/m128 and xmm1 for equality.
VEX.NDS.128.66.0F.WIG 74 /r VPCMPEQB xmm1, xmm2, xmm3 /m128	B	V/V	AVX	Compare packed bytes in xmm3/m128 and xmm2 for equality.
VEX.NDS.128.66.0F.WIG 75 /r VPCMPEQW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed words in xmm3/m128 and xmm2 for equality.
VEX.NDS.128.66.0F.WIG 76 /r VPCMPEQD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed doublewords in xmm3/m128 and xmm2 for equality.
VEX.NDS.128.66.0F38.WIG 29 /r VPCMPEQQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed quadwords in xmm3/m128 and xmm2 for equality.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare for equality of the packed bytes, words, doublewords, or quadwords in the first source operand and the second source operand. If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers.

The PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the PCMPEQW instruction compares the corresponding words in the destination and source operands; the PCMPEQD instruction compares the corresponding doublewords in the destination and source operands, and the PCMPEQQ instruction compares the corresponding quadwords in the destination and source operands.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
COMPARE_BYTES_EQUAL (SRC1, SRC2)
    IF SRC1[7:0] = SRC2[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
    IF SRC1[127:120] = SRC2[127:120]
    THEN DEST[127:120] ← FFH;
    ELSE DEST[127:120] ← 0; FI;

COMPARE_WORDS_EQUAL (SRC1, SRC2)
    IF SRC1[15:0] = SRC2[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
    IF SRC1[127:112] = SRC2[127:112]
    THEN DEST[127:112] ← FFFFH;
    ELSE DEST[127:112] ← 0; FI;
```

```
COMPARE_DWORDS_EQUAL (SRC1, SRC2)
    IF SRC1[31:0] = SRC2[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
    IF SRC1[127:96] = SRC2[127:96]
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 0; FI;

COMPARE_QWORDS_EQUAL (SRC1, SRC2)
    IF SRC1[63:0] = SRC2[63:0]
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0; FI;
    IF SRC1[127:64] = SRC2[127:64]
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0; FI;
```

VPCMPEQB (VEX.128 encoded version)
DEST[127:0] ←COMPARE_BYTES_EQUAL(SRC1,SRC2)
DEST[255:128] ← 0

PCMPEQB (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_BYTES_EQUAL(DEST,SRC)
DEST[255:128] (Unmodified)

VPCMPEQW (VEX.128 encoded version)
DEST[127:0] ←COMPARE_WORDS_EQUAL(SRC1,SRC2)
DEST[255:128] ← 0

PCMPEQW (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_WORDS_EQUAL(DEST,SRC)
DEST[255:128] (Unmodified)

VPCMPEQD (VEX.128 encoded version)
DEST[127:0] ←COMPARE_DWORDS_EQUAL(SRC1,SRC2)
DEST[255:128] ← 0

PCMPEQD (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_DWORDS_EQUAL(DEST,SRC)
DEST[255:128] (Unmodified)

VPCMPEQQ (VEX.128 encoded version)
DEST[127:0] ←COMPARE_QWORDS_EQUAL(SRC1,SRC2)

DEST[255:128] ← 0

PCMPEQQ (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_QWORDS_EQUAL(DEST,SRC)
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PCMPEQB __m128i _mm_cmpeq_epi8 (__m128i a, __m128i b)

PCMPEQW __m128i _mm_cmpeq_epi16 (__m128i a, __m128i b)

PCMPEQD __m128i _mm_cmpeq_epi32 (__m128i a, __m128i b)

PCMPEQQ __m128i _mm_cmpeq_epi64(__m128i a, __m128i b);

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PCMPESTRI - Packed Compare Explicit Length Strings, Return Index

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 61 /r ib PCMPESTRI xmm1, xmm2/m128, imm8	A	V/V	SSE4_2	Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX
VEX.128.66.0F3A.WIG 61 /r ib VPCMPESTRI xmm1, xmm2/m128, imm8	A	V/V	AVX	Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

The instruction compares data from two strings based on the control encoded in the imm8 byte (as described in *Section 3.1.2* of *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) generating an index stored to ECX. Each string is represented by two values. The first value is an XMM (or possibly m128 for the second operand) which contains the elements of the string (character data). The second value is stored in EAX (for xmm1) or EDX (for xmm2/m128) and represents the number of Bytes/Words which are valid for the respective xmm/m128 data. The length of each input is interpreted as being the absolute-value of the value in EAX (EDX). The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in EAX (EDX) is greater than 16 (8) or less than -16 (-8).

At this point the comparisons and aggregation described in section *Section 3.1.2* of *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A* are performed and the index of the first (or last, according to imm8[6]) set bit of IntRes2 is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner to supply the most relevant information.

CFlag – Reset if IntRes2 is equal to zero, set otherwise

ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise

SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise

OFlag – IntRes2[0]

AFlag – Reset

PFlag – Reset

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

See *PCMPESTRI Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B*.

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_cmpestri (__m128i a, int la, __m128i b, int lb, const int mode);

Intel C/C++ Compiler Intrinsic Equivalent for reading EFLAG Results

int _mm_cmpestria (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestric (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrio (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestris (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestriz (__m128i a, int la, __m128i b, int lb, const int mode);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

PCMPESTRM - Packed Compare Explicit Length Strings, Return Mask

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 60 /r ib PCMPESTRM xmm1, xmm2/m128, imm8	A	V/V	SSE4_2	Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in XMM0
VEX.128.66.0F3A.WIG 60 /r ib VPCMPESTRM xmm1, xmm2/m128, imm8	A	V/V	AVX	Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in XMM0

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

The instruction compares data from two strings based on the control encoded in the imm8 byte (as described in *Section 3.1.2* of *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) generating a mask stored to XMM0. Each string is represented by two values. The first value is an XMM (or possibly m128 for the second operand) which contains the elements of the string (character data). The second value is stored in EAX (for xmm1) or EDX (for xmm2/m128) and represents the number of Bytes/Words which are valid for the respective xmm/m128 data. The length of each input is interpreted as being the absolute-value of the value in EAX (EDX). The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in EAX (EDX) is greater than 16 (8) or less than -16 (-8).

At this point the comparisons and aggregation described in *Section 3.1.2* of *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A* are performed. As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner to supply the most relevant information

CFlag – Reset if IntRes2 is equal to zero, set otherwise

ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise

SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise

OFlag –IntRes2[0]

AFlag – Reset

PFlag – Reset

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 1, otherwise the instruction will #UD.

Operation

See *PCMPESTRM Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B*.

Intel C/C++ Compiler Intrinsic Equivalent

__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode);

Intel C/C++ Compiler Intrinsic Equivalent for reading EFLAG Results

int _mm_cmpestrma (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrmc (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrmo (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrms (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrmz (__m128i a, int la, __m128i b, int lb, const int mode);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

PCMPGTB/PCMPGTW/PCMPGTD/PCMPGTQ- Compare Packed Integers for Greater Than

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 64 /r PCMPGTB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed byte integers in xmm1 and xmm2/m128 for greater than.
66 0F 65 /r PCMPGTW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm1 and xmm2/m128 for greater than.
66 0F 66 /r PCMPGTD xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed doubleword integers in xmm1 and xmm2/m128 for greater than.
66 0F 38 37 /r PCMPGTQ xmm1, xmm2/m128	A	V/V	SSE4_2	Compare packed qwords in xmm2/m128 and xmm1 for greater than.
VEX.NDS.128.66.0F.WIG 64 /r VPCMPGTB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than.
VEX.NDS.128.66.0F.WIG 65 /r VPCMPGTW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm2 and xmm3/m128 for greater than.
VEX.NDS.128.66.0F.WIG 66 /r VPCMPGTD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed doubleword integers in xmm2 and xmm3/m128 for greater than.
VEX.NDS.128.66.0F38.WIG 37 /r VPCMPGTQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed qwords in xmm2 and xmm3/m128 for greater than.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD signed compare for the greater value of the packed byte, word, doubleword, or quadword integers in the first source operand and the second source operand. If a data element in the first source operand is greater than the corresponding date element in the second source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers.

The PCMPGTB instruction compares the corresponding signed byte integers in the first and second source operands; the PCMPGTW instruction compares the corresponding signed word integers in the first and second source operands; the PCMPGTD instruction compares the corresponding signed doubleword integers in the first and second source operands, and the PCMPGTQ instruction compares the corresponding signed qword integers in the first and second source operands.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
COMPARE_BYTES_GREATER (SRC1, SRC2)
    IF SRC1[7:0] > SRC2[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
    IF SRC1[127:120] > SRC2[127:120]
    THEN DEST[127:120] ← FFH;
    ELSE DEST[127:120] ← 0; FI;

COMPARE_WORDS_GREATER (SRC1, SRC2)
    IF SRC1[15:0] > SRC2[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
    IF SRC1[127:112] > SRC2[127:112]
    THEN DEST[127:112] ← FFFFH;
```

```
    ELSE DEST[127:112] ← 0; FI;

COMPARE_DWORDS_GREATER (SRC1, SRC2)
    IF SRC1[31:0] > SRC2[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
    IF SRC1[127:96] > SRC2[127:96]
    THEN DEST[127:96] ←FFFFFFFFH;
    ELSE DEST[127:96] ← 0; FI;

COMPARE_QWORDS_GREATER (SRC1, SRC2)
    IF SRC1[63:0] > SRC2[63:0]
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0; FI;
    IF SRC1[127:64] > SRC2[127:64]
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0; FI;
```

VPCMPGTB (VEX.128 encoded version)
DEST[127:0] ←COMPARE_BYTES_GREATER(SRC1,SRC2)
DEST[255:128] ← 0

PCMPGTB (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_BYTES_GREATER(DEST,SRC)
DEST[255:128] (Unmodified)

VPCMPGTW (VEX.128 encoded version)
DEST[127:0] ←COMPARE_WORDS_GREATER(SRC1,SRC2)
DEST[255:128] ← 0

PCMPGTW (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_WORDS_GREATER(DEST,SRC)
DEST[255:128] (Unmodified)

VPCMPGTD (VEX.128 encoded version)
DEST[127:0] ←COMPARE_DWORDS_GREATER(SRC1,SRC2)
DEST[255:128] ← 0

PCMPGTD (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_DWORDS_GREATER(DEST,SRC)
DEST[255:128] (Unmodified)

VPCMPGTQ (VEX.128 encoded version)
DEST[127:0] ←COMPARE_QWORDS_GREATER(SRC1,SRC2)
DEST[255:128] ← 0

PCMPGTQ (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_QWORDS_GREATER(DEST,SRC)
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PCMPGTB __m128i _mm_cmpgt_epi8 (__m128i a, __m128i b)

PCMPGTW __m128i _mm_cmpgt_epi16 (__m128i a, __m128i b)

PCMPGTD __m128i _mm_cmpgt_epi32 (__m128i a, __m128i b)

PCMPGTQ __m128i _mm_cmpgt_epi64(__m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PCMPISTRI - Packed Compare Implicit Length Strings, Return Index

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 63 /r ib PCMPISTRI xmm1, xmm2/m128, imm8	A	V/V	SSE4_2	Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX
VEX.128.66.0F3A.WIG 63 /r ib VPCMPISTRI xmm1, xmm2/m128, imm8	A	V/V	AVX	Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

The instruction compares data from two strings based on the control encoded in the imm8 byte (as described in *Section 3.1.2* of *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) generating an index stored to ECX. Each string is represented by a single value. The value is an XMM (or possibly m128 for the second operand) which contains the elements of the string (character data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.) At this point the comparisons and aggregation described in *Section 3.1.2* of *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A* are performed and the index of the first (or last, according to imm8[6]) set bit of IntRes2 is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner to supply the most relevant information.

CFlag – Reset if IntRes2 is equal to zero, set otherwise

ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise

SFlag – Set if any byte/word of xmm1 is null, reset otherwise

OFlag –IntRes2[0]

AFlag – Reset

PFlag – Reset

Note: In VEX.128 encoded version, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

See *PCMPISTRI Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B*.

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_cmpistri (__m128i a, __m128i b, const int mode);

Intel C/C++ Compiler Intrinsic Equivalent for reading EFLAG Results

int _mm_cmpistria (__m128i a, __m128i b, const int mode);

int _mm_cmpistric (__m128i a, __m128i b, const int mode);

int _mm_cmpistrio (__m128i a, __m128i b, const int mode);

int _mm_cmpistris (__m128i a, __m128i b, const int mode);

int _mm_cmpistriz (__m128i a, __m128i b, const int mode);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

PCMPISTRM - Packed Compare Implicit Length Strings, Return Mask

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 62 /r ib PCMPISTRM xmm1, xmm2/m128, imm8	A	V/V	SSE4_2	Perform a packed comparison of string data with implicit lengths, generating a Mask, and storing the result in XMM0
VEX.128.66.0F3A.WIG 62 /r ib VPCMPISTRM xmm1, xmm2/m128, imm8	A	V/V	AVX	Perform a packed comparison of string data with implicit lengths, generating a Mask, and storing the result in XMM0

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

The instruction compares data from two strings based on the control encoded in the imm8 byte (as described in *Section 3.1.2* of *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) generating a mask stored to XMM0. Each string is represented by a single value. The value is an XMM (or possibly m128 for the second operand) which contains the elements of the string (character data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

At this point the comparisons and aggregation described in *Section 3.1.2* of *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A* are performed. As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner to supply the most relevant information.

CFlag – Reset if IntRes2 is equal to zero, set otherwise

ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise

SFlag – Set if any byte/word of xmm1 is null, reset otherwise

OFlag – IntRes2[0]

AFlag – Reset

PFlag – Reset

Note: In VEX.128 encoded version, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

See *PCMPESTRM Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B*.

Intel C/C++ Compiler Intrinsic Equivalent

__m128i _mm_cmpistrm (__m128i a, __m128i b, const int mode)

Intel C/C++ Compiler Intrinsic Equivalent for reading EFLAG Results

int _mm_cmpistrma (__m128i a, __m128i b, const int mode);

int _mm_cmpistrmc (__m128i a, __m128i b, const int mode);

int _mm_cmpistrmo (__m128i a, __m128i b, const int mode);

int _mm_cmpistrms (__m128i a, __m128i b, const int mode);

int _mm_cmpistrmz (__m128i a, __m128i b, const int mode);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

VPERMILPD- Permute Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 0D /r VPERMILPD xmm1, xmm2, xmm3/m128	A	V/V	AVX	Permute double-precision floating-point values in xmm2 using controls from xmm3/mem and store result in xmm1
VEX.NDS.256.66.0F38.W0 0D /r VPERMILPD ymm1, ymm2, ymm3/m256	A	V/V	AVX	Permute double-precision floating-point values in ymm2 using controls from ymm3/mem and store result in ymm1
VEX.128.66.0F3A.W0 05 /r ib VPERMILPD xmm1, xmm2/m128, imm8	B	V/V	AVX	Permute double-precision floating-point values in xmm2/mem using controls from imm8
VEX.256.66.0F3A.W0 05 /r ib VPERMILPD ymm1, ymm2/m256, imm8	B	V/V	AVX	Permute double-precision floating-point values in ymm2/mem using controls from imm8

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Permute double-precision floating-point values in the first source operand (second operand) using 8-bit control fields in the low bytes of the second source operand (third operand) and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

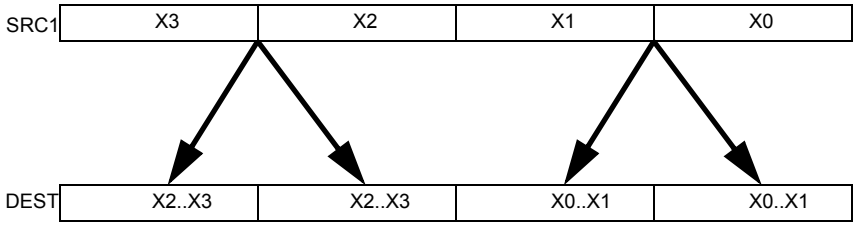


Figure 5-18. VPERMILPD operation

There is one control byte per destination double-precision element. Each control byte is aligned with the low 8 bits of the corresponding double-precision destination element. Each control byte contains a 1-bit select field (see Figure 5-19) that determines which of the source elements are selected. Source elements are restricted to lie in the same source 128-bit region as the destination.

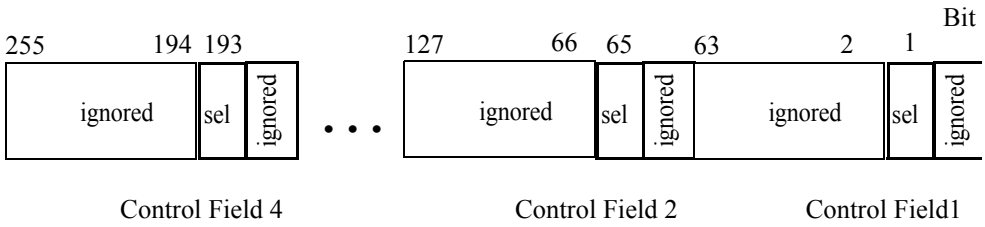


Figure 5-19. VPERMILPD Shuffle Control

(immediate control version)

Permute double-precision floating-point values in the first source operand (second operand) using two, 1-bit control fields in the low 2 bits of the 8-bit immediate and store results in the destination operand (first operand). The source operand is a YMM register or 256-bit memory location and the destination operand is a YMM register.

Note: For the VEX.128.66.0F3A 05 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Note: For the VEX.256.66.0F3A 05 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Operation

VPERMILPD (256-bit immediate version)

IF (imm8[0] = 0) THEN DEST[63:0]←SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0]←SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64]←SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64]←SRC1[127:64]
IF (imm8[2] = 0) THEN DEST[191:128]←SRC1[191:128]
IF (imm8[2] = 1) THEN DEST[191:128]←SRC1[255:192]
IF (imm8[3] = 0) THEN DEST[255:192]←SRC1[191:128]
IF (imm8[3] = 1) THEN DEST[255:192]←SRC1[255:192]

VPERMILPD (128-bit immediate version)

IF (imm8[0] = 0) THEN DEST[63:0]←SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0]←SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64]←SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64]←SRC1[127:64]
DEST[255:128]←0

VPERMILPD (256-bit variable version)

IF (SRC2[1] = 0) THEN DEST[63:0]←SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0]←SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64]←SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64]←SRC1[127:64]
IF (SRC2[129] = 0) THEN DEST[191:128]←SRC1[191:128]
IF (SRC2[129] = 1) THEN DEST[191:128]←SRC1[255:192]
IF (SRC2[193] = 0) THEN DEST[255:192]←SRC1[191:128]
IF (SRC2[193] = 1) THEN DEST[255:192]←SRC1[255:192]

VPERMILPD (128-bit variable version)

IF (SRC2[1] = 0) THEN DEST[63:0]←SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0]←SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64]←SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64]←SRC1[127:64]

DEST[255:128]←0

Intel C/C++ Compiler Intrinsic Equivalent

VPERMILPD __m128d _mm_permute_pd (__m128d a, int control)

VPERMILPD __m256d _mm256_permute_pd (__m256d a, int control)

VPERMILPD __m128d _mm_permutevar_pd (__m128d a, __m128i control);

VPERMILPD __m256d _mm256_permutevar_pd (__m256d a, __m256i control);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.W = 1

VPERMILPS- Permute Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 0C /r VPERMILPS xmm1, xmm2, xmm3/m128	A	V/V	AVX	Permute single-precision floating-point values in xmm2 using controls from xmm3/mem and store result in xmm1
VEX.128.66.0F3A.W0 04 /r ib VPERMILPS xmm1, xmm2/m128, imm8	B	V/V	AVX	Permute single-precision floating-point values in xmm2/mem using controls from imm8 and store result in xmm1
VEX.NDS.256.66.0F38.W0 0C /r VPERMILPS ymm1, ymm2, ymm3/m256	A	V/V	AVX	Permute single-precision floating-point values in ymm2 using controls from ymm3/mem and store result in ymm1
VEX.256.66.0F3A.W0 04 /r ib VPERMILPS ymm1, ymm2/m256, imm8	B	V/V	AVX	Permute single-precision floating-point values in ymm2/mem using controls from imm8 and store result in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

(variable control version)

Permute single-precision floating-point values in the first source operand (second operand) using 8-bit control fields in the low bytes of corresponding elements the shuffle control (third operand) and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

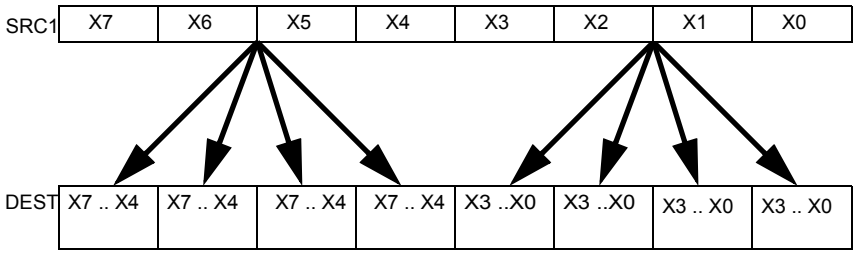


Figure 5-20. VPERMILPS Operation

There is one control byte per destination single-precision element. Each control byte is aligned with the low 8 bits of the corresponding single-precision destination element. Each control byte contains a 2-bit select field (see Figure 5-21) that determines which of the source elements are selected. Source elements are restricted to lie in the same source 128-bit region as the destination.

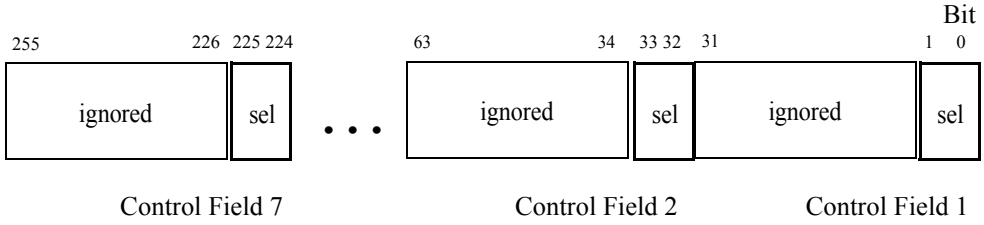


Figure 5-21. VPERMILPS Shuffle Control

(immediate control version)

Permute single-precision floating-point values in the first source operand (second operand) using four 2-bit control fields in the 8-bit immediate and store results in the destination operand (first operand). The source operand is a YMM register or 256-bit memory location and the destination operand is a YMM register. This is similar to a wider version of PSHUFD, just operating on single-precision floating-point values.

Note: For the VEX.128.66.0F3A 04 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Note: For the VEX.256.66.0F3A 04 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Operation

```
Select4(SRC, control) {
CASE (control[1:0]) OF
    0:    TMP ← SRC[31:0];
    1:    TMP ← SRC[63:32];
    2:    TMP ← SRC[95:64];
    3:    TMP ← SRC[127:96];
ESAC;
RETURN TMP
}
```

VPERMILPS (256-bit immediate version)

```
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] ← Select4(SRC1[255:128], imm8[5:4]);
DEST[255:224] ← Select4(SRC1[255:128], imm8[7:6]);
```

VPERMILPS (128-bit immediate version)

```
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[255:128]←0
```

VPERMILPS (256-bit variable version)

```
DEST[31:0] ← Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] ← Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] ← Select4(SRC1[127:0], SRC2[97:96]);
DEST[159:128] ← Select4(SRC1[255:128], SRC2[129:128]);
DEST[191:160] ← Select4(SRC1[255:128], SRC2[161:160]);
DEST[223:192] ← Select4(SRC1[255:128], SRC2[193:192]);
DEST[255:224] ← Select4(SRC1[255:128], SRC2[225:224]);
```

VPERMILPS (128-bit variable version)
DEST[31:0] ← Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] ← Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] ← Select4(SRC1[127:0], SRC2[97:96]);
DEST[255:128]←0

Intel C/C++ Compiler Intrinsic Equivalent

VPERM1LPS __m128 _mm_permute_ps (__m128 a, int control);

VPERM1LPS __m256 _mm256_permute_ps (__m256 a, int control);

VPERM1LPS __m128 _mm_permutevar_ps (__m128 a, __m128i control);

VPERM1LPS __m256 _mm256_permutevar_ps (__m256 a, __m256i control);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.W = 1

VPERM2F128- Permute Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 06 /r ib VPERM2F128 ymm1, ymm2, ymm3/m256, imm8	A	V/V	AVX	Permute 128-bit floating-point fields in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Permute 128 bit floating-point-containing fields from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

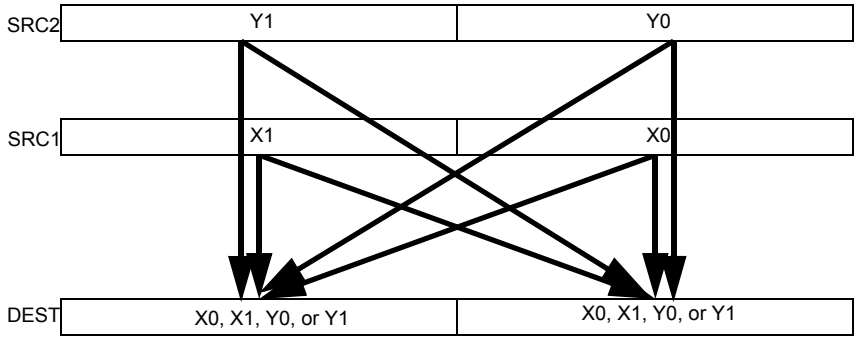


Figure 5-22. VPERM2F128 Operation

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed.

VEX.L must be 1, otherwise the instruction will #UD.

Operation

VPERM2F128

```
CASE IMM8[1:0] of
0: DEST[127:0] ← SRC1[127:0]
1: DEST[127:0] ← SRC1[255:128]
2: DEST[127:0] ← SRC2[127:0]
3: DEST[127:0] ← SRC2[255:128]
ESAC

CASE IMM8[5:4] of
0: DEST[255:128] ← SRC1[127:0]
1: DEST[255:128] ← SRC1[255:128]
2: DEST[255:128] ← SRC2[127:0]
3: DEST[255:128] ← SRC2[255:128]
ESAC
IF (imm8[3])
DEST[127:0] ← 0
FI

IF (imm8[7])
DEST[255:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2F128 __m256 _mm256_permute2f128_ps (__m256 a, __m256 b, int control)

VPERM2F128 __m256d _mm256_permute2f128_pd (__m256d a, __m256d b, int control)

VPERM2F128 __m256i _mm256_permute2f128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0
	If VEX.W = 1

PEXTRB/PEXTRW/PEXTRD/PEXTRQ- Extract Integer

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 14 /r ib PEXTRB reg/m8, xmm2, imm8	A	V/V	SSE4_1	Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.
66 0F C5 /r ib PEXTRW reg, xmm1, imm8	B	V/V	SSE2	Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. The upper bits of r64/r32 is filled with zeros.
66 0F 3A 15 /r ib PEXTRW reg/m16, xmm2, imm8	A	V/V	SSE4_1	Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.
66 0F 3A 16 /r ib PEXTRD r32/m32, xmm2, imm8	A	V/V	SSE4_1	Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.
66 REX.W 0F 3A 16 /r ib PEXTRQ r64/m64, xmm2, imm8	A	V/NE	SSE4_1	Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.
VEX.128.66.0F3A.W0 14 /r ib VPEXTRB reg/m8, xmm2, imm8	A	V/V	AVX	Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F.W0 C5 /r ib VPEXTRW reg, xmm1, imm8	B	V/V	AVX	Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F3A.W0 15 /r ib VPEXTRW reg/m16, xmm2, imm8	A	V/V	AVX	Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 16 /r ib VPEXTRD r32/m32, xmm2, imm8	A	V/V	AVX	Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.
VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ r64/m64, xmm2, imm8	A	V/INV	AVX	Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
B	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Extract a byte/word/dword/qword integer value from the source XMM register at a byte/word/dword/qword offset determined from imm8[3:0]. The destination can be a register or byte/word/dword/qword memory location. If the destination is a register, the upper bits of the register are zero extended.

In legacy non-VEX encoded version and if the destination operand is a register, the default operand size in 64-bit mode for PEXTRB/PEXTRW/PEXTRD is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros. PEXTRQ is not encodable in non-64-bit modes and requires REX.W in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRB/VPEXTRW/VPEXTRD is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros. Attempt to execute VPEXTRQ in non-64-bit mode will cause #UD.

Operation

(V)PEXTRTD/(V)PEXTRQ

IF (64-Bit Mode and 64-bit dest operand)

THEN

```
    Src_Offset ← Imm8[0]
    r64/m64 ←(Src >> Src_Offset * 64)
ELSE
    Src_Offset ← Imm8[1:0]
    r32/m32 ← ((Src >> Src_Offset *32) AND 0FFFFFFFFh);
FI
```

(V)PEXTRW (dest=m16)

```
SRC_Offset ← Imm8[2:0]
Mem16 ← (Src >> Src_Offset*16)
```

(V)PEXTRW (dest=reg)

```
IF (64-Bit Mode )
THEN
    SRC_Offset ← Imm8[2:0]
    DEST[15:0] ← ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[63:16] ←    ZERO_FILL;
ELSE
    SRC_Offset ← Imm8[2:0]
    DEST[15:0] ← ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[31:16] ←    ZERO_FILL;
FI
```

(V)PEXTRB (dest=m8)

```
SRC_Offset ← Imm8[3:0]
Mem8 ← (Src >> Src_Offset*8)
```

(V)PEXTRB (dest=reg)

```
IF (64-Bit Mode )
THEN
    SRC_Offset ← Imm8[3:0]
    DEST[7:0] ← ((Src >> Src_Offset*8) AND 0FFh)
    DEST[63:8] ←    ZERO_FILL;
ELSE
    SRC_Offset ←. Imm8[3:0];
    DEST[7:0] ← ((Src >> Src_Offset*8) AND 0FFh);
    DEST[31:8] ←    ZERO_FILL;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PEXTRB int _mm_extract_epi8 (__m128i src, const int ndx);

PEXTRW int _mm_extract_epi16 (__m128i src, int ndx);

PEXTRD int _mm_extract_epi32 (__m128i src, const int ndx);

PEXTRQ __int64 _mm_extract_epi64 (__m128i src, const int ndx);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.
	If VPEXTRQ in non-64-bit mode, VEX.W=1

PHADDW/PHADDD - Packed Horizontal Add

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 01 /r PHADDW xmm1, xmm2/m128	A	V/V	SSSE3	Add 16-bit signed integers horizontally, pack to xmm1.
66 0F 38 02 /r PHADDD xmm1, xmm2/m128	A	V/V	SSSE3	Add 32-bit signed integers horizontally, pack to xmm1.
VEX.NDS.128.66.0F38.WIG 01 /r VPHADDW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add 16-bit signed integers horizontally, pack to xmm1.
VEX.NDS.128.66.0F38.WIG 02 /r VPHADDD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add 32-bit signed integers horizontally, pack to xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PHADDW adds two adjacent 16-bit signed integers horizontally from the second source operand and the first source operand and packs the 16-bit signed results to the destination operand. PHADDD adds two adjacent 32-bit signed integers horizontally from the second source operand and the first source operand and packs the 32-bit signed results to the destination operand. The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPHADDW (VEX.128 encoded version)

DEST[15:0] ← SRC1[31:16] + SRC1[15:0]
DEST[31:16] ← SRC1[63:48] + SRC1[47:32]
DEST[47:32] ← SRC1[95:80] + SRC1[79:64]
DEST[63:48] ← SRC1[127:112] + SRC1[111:96]
DEST[79:64] ← SRC2[31:16] + SRC2[15:0]
DEST[95:80] ← SRC2[63:48] + SRC2[47:32]
DEST[111:96] ← SRC2[95:80] + SRC2[79:64]
DEST[127:112] ← SRC2[127:112] + SRC2[111:96]
DEST[255:128] ← 0

VPHADDD (VEX.128 encoded version)

DEST[31-0] ← SRC1[63-32] + SRC1[31-0]
DEST[63-32] ← SRC1[127-96] + SRC1[95-64]
DEST[95-64] ← SRC2[63-32] + SRC2[31-0]
DEST[127-96] ← SRC2[127-96] + SRC2[95-64]
DEST[255:128] ← 0

PHADDW (128-bit Legacy SSE version)

DEST[15:0] ← DEST[31:16] + DEST[15:0]
DEST[31:16] ← DEST[63:48] + DEST[47:32]
DEST[47:32] ← DEST[95:80] + DEST[79:64]
DEST[63:48] ← DEST[127:112] + DEST[111:96]
DEST[79:64] ← SRC[31:16] + SRC[15:0]
DEST[95:80] ← SRC[63:48] + SRC[47:32]
DEST[111:96] ← SRC[95:80] + SRC[79:64]
DEST[127:112] ← SRC[127:112] + SRC[111:96]
DEST[255:128] (Unmodified)

PHADDD (128-bit Legacy SSE version)

DEST[31-0] ← DEST[63-32] + DEST[31-0]
DEST[63-32] ← DEST[127-96] + DEST[95-64]
DEST[95-64] ← SRC[63-32] + SRC[31-0]
DEST[127-96] ← SRC[127-96] + SRC[95-64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PHADDW __m128i _mm_hadd_epi16 (__m128i a, __m128i b)

PHADDD __m128i _mm_hadd_epi32 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PHADDSW - Packed Horizontal Add with Saturation

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 03 /r PHADDSW xmm1, xmm2/m128	A	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to xmm1.
VEX.NDS.128.66.0F38.WIG 03 /r VPHADDSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add 16-bit signed integers horizontally, pack saturated integers to xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PHADDSW adds two adjacent signed 16-bit integers horizontally from the second source and first source operands and saturates the signed results; packs the signed, saturated 16-bit results to the destination operand. The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPHADDSW (VEX.128 encoded version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
```

DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[255:128] ← 0

PHADDSW (128-bit Legacy SSE version)
DEST[15:0]= SaturateToSignedWord(DEST[31:16] + DEST[15:0])
DEST[31:16] = SaturateToSignedWord(DEST[63:48] + DEST[47:32])
DEST[47:32] = SaturateToSignedWord(DEST[95:80] + DEST[79:64])
DEST[63:48] = SaturateToSignedWord(DEST[127:112] + DEST[111:96])
DEST[79:64] = SaturateToSignedWord(SRC[31:16] + SRC[15:0])
DEST[95:80] = SaturateToSignedWord(SRC[63:48] + SRC[47:32])
DEST[111:96] = SaturateToSignedWord(SRC[95:80] + SRC[79:64])
DEST[127:112] = SaturateToSignedWord(SRC[127:112] + SRC[111:96])
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PHADDSW __m128i _mm_hadds_epi16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PHMINPOSUW - Horizontal Minimum and Position

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 41 /r PHMINPOSUW xmm1, xmm2/m128	A	V/V	SSE4_1	Find the minimum unsigned word in xmm2/m128 and place its value in the low word of xmm1 and its index in the second-lowest word of xmm1
VEX.128.66.0F38.WIG 41 /r VPHMINPOSUW xmm1, xmm2/m128	A	V/V	AVX	Find the minimum unsigned word in xmm2/m128 and place its value in the low word of xmm1 and its index in the second-lowest word of xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Determine the minimum unsigned word value in the source operand and place the unsigned word in the low word (bits 0-15) of the destination operand. The word index of the minimum value is stored in bits 16-18 of the destination operand. The remaining upper bits of the destination are set to zero.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPHMINPOSUW (VEX.128 encoded version)

INDEX ← 0
MIN ← SRC[15:0]
IF (SRC[31:16] < MIN) THEN INDEX ← 1; MIN ← SRC[31:16]
IF (SRC[47:32] < MIN) THEN INDEX ← 2; MIN ← SRC[47:32]
* Repeat operation for words 3 through 6

IF (SRC[127:112] < MIN) THEN INDEX ← 7; MIN ← SRC[127:112]
DEST[15:0] ← MIN
DEST[18:16] ← INDEX
DEST[127:19] ← 00000000000000000000000000000H
DEST[255:128] ← 0

PHMINPOSUW (128-bit Legacy SSE version)
INDEX ← 0
MIN ← SRC[15:0]
IF (SRC[31:16] < MIN) THEN INDEX ← 1; MIN ← SRC[31:16]
IF (SRC[47:32] < MIN) THEN INDEX ← 2; MIN ← SRC[47:32]
* Repeat operation for words 3 through 6
IF (SRC[127:112] < MIN) THEN INDEX ← 7; MIN ← SRC[127:112]
DEST[15:0] ← MIN
DEST[18:16] ← INDEX
DEST[127:19] ← 00000000000000000000000000000H
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PHMINPOSUW __m128i _mm_minpos_epu16(__m128i packed_words)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

PHSUBW/PHSUBD - Packed Horizontal Subtract

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 05 /r PHSUBW xmm1, xmm2/m128	A	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to xmm1.
66 0F 38 06 /r PHSUBD xmm1, xmm2/m128	A	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to xmm1.
VEX.NDS.128.66.0F38.WIG 05 /r VPHSUBW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract 16-bit signed integers horizontally, pack to xmm1.
VEX.NDS.128.66.0F38.WIG 06 /r VPHSUBD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract 32-bit signed integers horizontally, pack to xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PHSUBW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the second source operand and destination operands, and packs the signed 16-bit results to the destination operand. PHSUBD performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant doubleword of each pair, and packs the signed 32-bit result to the destination operand.

The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPHSUBW (VEX.128 encoded version)

DEST[15:0] ← SRC1[15:0] - SRC1[31:16]
DEST[31:16] ← SRC1[47:32] - SRC1[63:48]
DEST[47:32] ← SRC1[79:64] - SRC1[95:80]
DEST[63:48] ← SRC1[111:96] - SRC1[127:112]
DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[255:128] ← 0

VPHSUBD (VEX.128 encoded version)

DEST[31-0] ← SRC1[31-0] - SRC1[63-32]
DEST[63-32] ← SRC1[95-64] - SRC1[127-96]
DEST[95-64] ← SRC2[31-0] - SRC2[63-32]
DEST[127-96] ← SRC2[95-64] - SRC2[127-96]
DEST[255:128] ← 0

PHSUBW (128-bit Legacy SSE version)

DEST[15:0] ← DEST[15:0] - DEST[31:16]
DEST[31:16] ← DEST[47:32] - DEST[63:48]
DEST[47:32] ← DEST[79:64] - DEST[95:80]
DEST[63:48] ← DEST[111:96] - DEST[127:112]
DEST[79:64] ← SRC[15:0] - SRC[31:16]
DEST[95:80] ← SRC[47:32] - SRC[63:48]
DEST[111:96] ← SRC[79:64] - SRC[95:80]
DEST[127:112] ← SRC[111:96] - SRC[127:112]
DEST[255:128] (Unmodified)

PHSUBD (128-bit Legacy SSE version)

DEST[31-0] ← DEST[31-0] - DEST[63-32]
DEST[63-32] ← DEST[95-64] - DEST[127-96]
DEST[95-64] ← SRC[31-0] - SRC[63-32]
DEST[127-96] ← SRC[95-64] - SRC[127-96]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PHSUBW __m128i _mm_hsub_epi16 (__m128i a, __m128i b)

PHSUBD __m128i _mm_hsub_epi32 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PHSUBSW - Packed Horizontal Subtract with Saturation

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 07 /r PHSUBSW xmm1, xmm2/m128	A	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to xmm1
VEX.NDS.128.66.0F38.WIG 07 /r VPHSUBSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract 16-bit signed integer horizontally, pack saturated integers to xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PHSUBSW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the second source and first source operands. The signed, saturated 16-bit results are packed to the destination operand. The destination and first source operand are XMM registers. The second operand can be an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPHSUBSW (VEX.128 encoded version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
```

DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[255:128] ← 0

PHSUBSW (128-bit Legacy SSE version)
DEST[15:0]= SaturateToSignedWord(DEST[15:0] - DEST[31:16])
DEST[31:16] = SaturateToSignedWord(DEST[47:32] - DEST[63:48])
DEST[47:32] = SaturateToSignedWord(DEST[79:64]) - DEST[95:80]
DEST[63:48] = SaturateToSignedWord(DEST[111:96] - DEST[127:112])
DEST[79:64] = SaturateToSignedWord(SRC[15:0] - SRC[31:16])
DEST[95:80] = SaturateToSignedWord(SRC[47:32] - SRC[63:48])
DEST[111:96] = SaturateToSignedWord(SRC[79:64] - SRC[95:80])
DEST[127:112] = SaturateToSignedWord(SRC[SRC[111:96] - 127:112])
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PHSUBSW __m128i _mm_hsubs_epi16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PINSRB/PINSRW/PINSRD/PINSRQ- Insert Integer

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 20 /r ib PINSRB xmm1, r32/m8, imm8	A	V/V	SSE4_1	Insert a byte integer value from r32/m8 into xmm1 at the byte offset in imm8
66 0F C4 /r ib PINSRW xmm1, r32/m16, imm8	A	V/V	SSE2	Insert a word integer value from r32/m16 into xmm1 at the word offset in imm8
66 0F 3A 22 /r ib PINSRD xmm1, r32/m32, imm8	A	V/V	SSE4_1	Insert a dword integer value from r32/m32 into xmm1 at the dword offset in imm8
66 REX.W 0F 3A 22 /r ib PINSRQ xmm1, r64/m64, imm8	A	V/N.E.	SSE4_1	Insert a qword integer value from r64/m64 into xmm1 at the qword offset in imm8
VEX.NDS.128.66.0F3A.W0 20 /r ib VPINSRB xmm1, xmm2, r32/m8, imm8	B	V/V	AVX	Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8
VEX.NDS.128.66.0F.W0 C4 /r ib VPINSRW xmm1, xmm2, r32/m16, imm8	B	V/V	AVX	Insert a word integer value from r32/m16 and rest from xmm2 into xmm1 at the word offset in imm8
VEX.NDS.128.66.0F3A.W0 22 /r ib VPINSRD xmm1, xmm2, r32/m32, imm8	B	V/V	AVX	Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8
VEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ xmm1, xmm2, r64/m64, imm8	B	V/INV	AVX	Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Copies a byte/word/dword/qword from the second source operand and inserts it into the destination operand at the byte/word/dword/qword offset specified with the immediate operand (third operand). The other bytes/words/dwords/qwords in the destination register are copied from the first source operand. The byte select is specified by the 4/3/2/1 least-significant bits of the immediate.

The first source operand and destination operands are XMM registers. The second source operand is a r32 register or an 8-/16-/32-bit memory location for byte/word/dword operation. Qword operation is supported only in 64-bit mode, the second operand is a 64-bit register or memory location. REX.W is required to encode PINSRQ in 64-bit mode, PINSRQ is not encodable in non-64-bit modes.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD. Attempt to execute VPINSRQ in non-64-bit mode will cause #UD.

Operation

```
write_q_element(position, val, src)
{
TEMP ← SRC
CASE (position)
0: TEMP[63:0] ← val
1: TEMP[127:64] ← val
ESAC
return TEMP
}

write_d_element(position, val, src)
{
TEMP ← SRC
CASE (position)
0: TEMP[31:0] ← val
1: TEMP[63:32] ← val
2: TEMP[95:64] ← val
3: TEMP[127:96] ← val
ESAC
```

return TEMP
}

write_w_element(position, val, src)
{
TEMP ← SRC
CASE (position)
0: TEMP[15:0] ← val
1: TEMP[31:16] ← val
2: TEMP[47:32] ← val
3: TEMP[63:48] ← val
4: TEMP[79:64] ← val
5: TEMP[95:80] ← val
6: TEMP[111:96] ← val
7: TEMP[127:112] ← val
ESAC
return TEMP
}

write_b_element(position, val, src)
{
TEMP ← SRC
CASE (position)
0: TEMP[7:0] ← val
1: TEMP[15:8] ← val
2: TEMP[23:16] ← val
3: TEMP[31:24] ← val
4: TEMP[39:32] ← val
5: TEMP[47:40] ← val
6: TEMP[55:48] ← val
7: TEMP[63:56] ← val
8: TEMP[71:64] ← val
9: TEMP[79:72] ← val
10: TEMP[87:80] ← val
11: TEMP[95:88] ← val
12: TEMP[103:96] ← val
13: TEMP[111:104] ← val
14: TEMP[119:112] ← val
15: TEMP[127:120] ← val
ESAC
return TEMP
}
VPINSRQ (VEX.128 encoded version)

SEL ← imm8[0]
DEST[127:0] ← write_q_element(SEL, SRC2, SRC1)
DEST[255:128] ← 0

VPINSRD (VEX.128 encoded version)
SEL ← imm8[1:0]
DEST[127:0] ← write_d_element(SEL, SRC2, SRC1)
DEST[255:128] ← 0

VPINSRW (VEX.128 encoded version)
SEL ← imm8[2:0]
DEST[127:0] ← write_w_element(SEL, SRC2, SRC1)
DEST[255:128] ← 0

VPINSRB (VEX.128 encoded version)
SEL ← imm8[3:0]
DEST[127:0] ← write_b_element(SEL, SRC2, SRC1)
DEST[255:128] ← 0

PINSRQ (Legacy SSE version)
SEL ← imm8[0]
DEST[127:0] ← write_q_element(SEL, SRC, DEST)
DEST[255:128] (Unmodified)

PINSRD (Legacy SSE version)
SEL ← imm8[1:0]
DEST[127:0] ← write_d_element(SEL, SRC, DEST)
DEST[255:128] (Unmodified)

PINSRW (Legacy SSE version)
SEL ← imm8[2:0]
DEST[127:0] ← write_w_element(SEL, SRC, DEST)
DEST[255:128] (Unmodified)

PINSRB (Legacy SSE version)
SEL ← imm8[3:0]
DEST[127:0] ← write_b_element(SEL, SRC, DEST)
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PINSRB __m128i _mm_insert_epi8 (__m128i s1, int s2, const int ndx);

PINSRW __m128i _mm_insert_epi16 (__m128i a, int b, int imm)

PINSRD __m128i _mm_insert_epi32 (__m128i s2, int s, const int ndx);

PINSRQ __m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VINSRQ in non-64-bit mode with VEX.W=1.

PMADDWD- Multiply and Add Packed Integers

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F F5 /r PMADDWD xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed word integers in xmm1 by the packed word integers in xmm2/m128, add adjacent doubleword results, and store in xmm1.
VEX.NDS.128.66.0F.WIG F5 /r VPMADDWD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15:0) and (31-16) in the second source and first source operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. The second source operand is an XMM register or a 128-bit memory location.

The first source and destination operands are XMM registers. The PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPMADDWD (VEX.128 encoded version)

DEST[31:0] ← (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] ← (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] ← (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] ← (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[255:128] ← 0

PMADDWD (128-bit Legacy SSE version)

DEST[31:0] ← (DEST[15:0] * SRC[15:0]) + (DEST[31:16] * SRC[31:16])
DEST[63:32] ← (DEST[47:32] * SRC[47:32]) + (DEST[63:48] * SRC[63:48])
DEST[95:64] ← (DEST[79:64] * SRC[79:64]) + (DEST[95:80] * SRC[95:80])
DEST[127:96] ← (DEST[111:96] * SRC[111:96]) + (DEST[127:112] * SRC[127:112])
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMADDWD __m128i _mm_madd_epi16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMADDUBSW- Multiply and Add Packed Integers

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 04 /r PMADDUBSW xmm1, xmm2/m128	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1.
VEX.NDS.128.66.0F38.WIG 04 /r VPMADDUBSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PMADDUBSW multiplies vertically each unsigned byte of the first source operand with the corresponding signed byte of the second source operand, producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7:0) in the first source and second source operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15:8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15:0). The same operation is performed on the other pairs of adjacent bytes. The second source operand can be an XMM register or 128-bit memory location. The first source operand and destination operands are XMM registers.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPMADDUBSW (VEX.128 encoded version)

DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])

// Repeat operation for 2nd through 7th word
DEST[127:112] ← SaturateToSignedWord(SRC2[127:120]*SRC1[127:120]+ SRC2[119:112]* SRC1[119:112])
DEST[255:128] ← 0

PMADDUBSW (128-bit Legacy SSE version)
DEST[15:0] ← SaturateToSignedWord(SRC[15:8]* DEST[15:8]+SRC[7:0]*DEST[7:0]);
// Repeat operation for 2nd through 7th word
DEST[127:112] ← SaturateToSignedWord(SRC[127:120]*DEST[127:120]+ SRC[119:112]* DEST[119:112]);
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMADDUBSW __m128i _mm_maddubs_epi16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMAXSB/PMAXSW/PMAXSD- Maximum of Packed Signed Integers

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3C /r PMAXSB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F EE /r PMAXSW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
66 0F 38 3D /r PMAXSD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3C /r VPMAXSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F.WIG EE /r VPMAXSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3D /r VPMAXSD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed signed byte, word, or dword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand. The first source and destination operand is an XMM register; The second source operand is an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMAXSB (128-bit Legacy SSE version)

```
    IF DEST[7:0] >SRC[7:0] THEN
        DEST[7:0] ← DEST[7:0];
    ELSE
        DEST[15:0] ← SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] >SRC[127:120] THEN
        DEST[127:120] ← DEST[127:120];
    ELSE
        DEST[127:120] ← SRC[127:120]; FI;
DEST[255:128] (Unmodified)
```

VPMAXSB (VEX.128 encoded version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[7:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] >SRC2[127:120] THEN
        DEST[127:120] ← SRC1[127:120];
    ELSE
        DEST[127:120] ← SRC2[127:120]; FI;
DEST[255:128] ← 0
```

PMAXSW (128-bit Legacy SSE version)

```
    IF DEST[15:0] >SRC[15:0] THEN
        DEST[15:0] ← DEST[15:0];
    ELSE
        DEST[15:0] ← SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] >SRC[127:112] THEN
        DEST[127:112] ← DEST[127:112];
    ELSE
        DEST[127:112] ← SRC[127:112]; FI;
DEST[255:128] (Unmodified)
```

VPMAXSW (VEX.128 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] >SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[255:128] ← 0
```

PMAXSD (128-bit Legacy SSE version)

```
    IF DEST[31:0] >SRC[31:0] THEN
        DEST[31:0] ← DEST[31:0];
    ELSE
        DEST[31:0] ← SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:95] >SRC[127:95] THEN
        DEST[127:95] ← DEST[127:95];
    ELSE
        DEST[127:95] ← SRC[127:95]; FI;
DEST[255:128] (Unmodified)
```

VPMAXSD (VEX.128 encoded version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] ← SRC1[31:0];
    ELSE
        DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:95] > SRC2[127:95] THEN
        DEST[127:95] ← SRC1[127:95];
```

```
    ELSE
        DEST[127:95] ← SRC2[127:95]; FI;
DEST[255:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMAXSB __m128i _mm_max_epi8 (__m128i a, __m128i b);

PMAXSW __m128i _mm_max_epi16 (__m128i a, __m128i b)

PMAXSD __m128i _mm_max_epi32 (__m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMAXUB/PMAXUW/PMAXUD- Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F DE /r PMAXUB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F 38 3E/r PMAXUW xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
66 0F 38 3F /r PMAXUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F.WIG DE /r VPMAXUB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3E/r VPMAXUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and store maximum packed values in xmm1.
VEX.NDS.128.66.0F38.WIG 3F /r VPMAXUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed unsigned byte, word, or dword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand. The first source and destination operand is an XMM register; The second source operand is an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMAXUB (128-bit Legacy SSE version)

```
    IF DEST[7:0] >SRC[7:0] THEN
        DEST[7:0] ← DEST[7:0];
    ELSE
        DEST[15:0] ← SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] >SRC[127:120] THEN
        DEST[127:120] ← DEST[127:120];
    ELSE
        DEST[127:120] ← SRC[127:120]; FI;
DEST[255:128] (Unmodified)
```

VPMAXUB (VEX.128 encoded version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[7:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] >SRC2[127:120] THEN
        DEST[127:120] ← SRC1[127:120];
    ELSE
        DEST[127:120] ← SRC2[127:120]; FI;
DEST[255:128] ← 0
```

PMAXUW (128-bit Legacy SSE version)

```
    IF DEST[15:0] >SRC[15:0] THEN
        DEST[15:0] ← DEST[15:0];
    ELSE
        DEST[15:0] ← SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] >SRC[127:112] THEN
        DEST[127:112] ← DEST[127:112];
    ELSE
        DEST[127:112] ← SRC[127:112]; FI;
DEST[255:128] (Unmodified)
```

VPMAXUW (VEX.128 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] >SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[255:128] ← 0
```

PMAXUD (128-bit Legacy SSE version)

```
    IF DEST[31:0] >SRC[31:0] THEN
        DEST[31:0] ← DEST[31:0];
    ELSE
        DEST[31:0] ← SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:95] >SRC[127:95] THEN
        DEST[127:95] ← DEST[127:95];
    ELSE
        DEST[127:95] ← SRC[127:95]; FI;
DEST[255:128] (Unmodified)
```

VPMAXUD (VEX.128 encoded version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] ← SRC1[31:0];
    ELSE
        DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:95] > SRC2[127:95] THEN
        DEST[127:95] ← SRC1[127:95];
```

```
    ELSE
        DEST[127:95] ← SRC2[127:95]; FI;
DEST[255:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMAXUB __m128i _mm_max_epu8 (__m128i a, __m128i b);

PMAXUW __m128i _mm_max_epu16 (__m128i a, __m128i b)

PMAXUD __m128i _mm_max_epu32 (__m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMINSB/PMINSW/PMINSD- Minimum of Packed Signed Integers

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 38 /r PMINSB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F EA /r PMINSW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
66 0F 38 39 /r PMINSD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 38 /r VPMINSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F.WIG EA /r VPMINSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 39 /r VPMINSD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed signed byte, word, or dword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand. The first source and destination operand is an XMM register; The second source operand is an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMINSB (128-bit Legacy SSE version)

```
    IF DEST[7:0] < SRC[7:0] THEN
        DEST[7:0] ← DEST[7:0];
    ELSE
        DEST[15:0] ← SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] < SRC[127:120] THEN
        DEST[127:120] ← DEST[127:120];
    ELSE
        DEST[127:120] ← SRC[127:120]; FI;
DEST[255:128] (Unmodified)
```

VPMINSB (VEX.128 encoded version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[7:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] < SRC2[127:120] THEN
        DEST[127:120] ← SRC1[127:120];
    ELSE
        DEST[127:120] ← SRC2[127:120]; FI;
DEST[255:128] ← 0
```

PMINSW (128-bit Legacy SSE version)

```
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] ← DEST[15:0];
    ELSE
        DEST[15:0] ← SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] < SRC[127:112] THEN
        DEST[127:112] ← DEST[127:112];
    ELSE
        DEST[127:112] ← SRC[127:112]; FI;
DEST[255:128] (Unmodified)
```

VPMINSW (VEX.128 encoded version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] < SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[255:128] ← 0
```

PMINSD (128-bit Legacy SSE version)

```
    IF DEST[31:0] < SRC[31:0] THEN
        DEST[31:0] ← DEST[31:0];
    ELSE
        DEST[31:0] ← SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:95] < SRC[127:95] THEN
        DEST[127:95] ← DEST[127:95];
    ELSE
        DEST[127:95] ← SRC[127:95]; FI;
DEST[255:128] (Unmodified)
```

VPMINSD (VEX.128 encoded version)

```
    IF SRC1[31:0] < SRC2[31:0] THEN
        DEST[31:0] ← SRC1[31:0];
    ELSE
        DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:95] < SRC2[127:95] THEN
        DEST[127:95] ← SRC1[127:95];
```

```
    ELSE
        DEST[127:95] ← SRC2[127:95]; FI;
DEST[255:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMINSB __m128i _mm_min_epi8 (__m128i a, __m128i b);

PMINSW __m128i _mm_min_epi16 (__m128i a, __m128i b)

PMINSD __m128i _mm_min_epi32 (__m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMINUB/PMINUW/PMINUD- Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F DA /r PMINUB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F 38 3A/r PMINUW xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
66 0F 38 3B /r PMINUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F.WIG DA /r VPMINUB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3A/r VPMINUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3B /r VPMINUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed unsigned byte, word, or dword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand. The first source and destination operand is an XMM register; The second source operand is an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMINUB (128-bit Legacy SSE version)

```
PMINUB instruction for 128-bit operands:
    IF DEST[7:0] < SRC[7:0] THEN
        DEST[7:0] ← DEST[7:0];
    ELSE
        DEST[15:0] ← SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] < SRC[127:120] THEN
        DEST[127:120] ← DEST[127:120];
    ELSE
        DEST[127:120] ← SRC[127:120]; FI;
DEST[255:128] (Unmodified)
```

VPMINUB (VEX.128 encoded version)

```
VPMINUB instruction for 128-bit operands:
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[7:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] < SRC2[127:120] THEN
        DEST[127:120] ← SRC1[127:120];
    ELSE
        DEST[127:120] ← SRC2[127:120]; FI;
DEST[255:128] ← 0
```

PMINUW (128-bit Legacy SSE version)

```
PMINUW instruction for 128-bit operands:
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] ← DEST[15:0];
    ELSE
        DEST[15:0] ← SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] < SRC[127:112] THEN
        DEST[127:112] ← DEST[127:112];
    ELSE
        DEST[127:112] ← SRC[127:112]; FI;
DEST[255:128] (Unmodified)
```

VPMINUW (VEX.128 encoded version)

```
VPMINUW instruction for 128-bit operands:
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] < SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[255:128] ← 0
```

PMINUD (128-bit Legacy SSE version)

```
PMINUD instruction for 128-bit operands:
    IF DEST[31:0] < SRC[31:0] THEN
        DEST[31:0] ← DEST[31:0];
    ELSE
        DEST[31:0] ← SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:95] < SRC[127:95] THEN
        DEST[127:95] ← DEST[127:95];
    ELSE
        DEST[127:95] ← SRC[127:95]; FI;
DEST[255:128] (Unmodified)
```

VPMINUD (VEX.128 encoded version)

```
VPMINUD instruction for 128-bit operands:
    IF SRC1[31:0] < SRC2[31:0] THEN
```

```
        DEST[31:0] ← SRC1[31:0];
    ELSE
        DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:95] < SRC2[127:95] THEN
        DEST[127:95] ← SRC1[127:95];
    ELSE
        DEST[127:95] ← SRC2[127:95]; FI;
DEST[255:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMINUB __m128i _mm_min_epu8 (__m128i a, __m128i b)

PMINUW __m128i _mm_min_epu16 (__m128i a, __m128i b);

PMINUD __m128i _mm_min_epu32 (__m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMOVMSKB- Move Byte Mask

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F D7 /r PMOVMSKB reg, xmm1	A	V/V	SSE2	Move a byte mask of xmm1 to reg. The upper bits of r32 or r64 are filled with zeros.
VEX.128.66.0F.WIG D7 /r VPMOVMSKB reg, xmm1	A	V/V	AVX	Move a byte mask of xmm1 to reg. The upper bits of r32 or r64 are filled with zeros.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:reg (r)	NA	NA

Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low byte or word of the destination operand (first operand). The source operand is an XMM register; the destination operand is a general-purpose register. The byte mask is 16-bits.

The destination operand is a general-purpose register. In 64-bit mode, the default operand size of the destination operand is 64 bits. The upper bits above bit 15 are filled with zeros. REX.W is ignored

VEX.128 encodings are valid but identical in function. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

(V)PMOVMSKB instruction with 128-bit source operand and r32:
r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r32[15] ← SRC[127];
r32[31:16] ← ZERO_FILL;

(V)PMOVMSKB instruction with 128-bit source operand and r64:
r64[0] ← SRC[7];
r64[1] ← SRC[15];

```
(* Repeat operation for bytes 2 through 14 *)
r64[15] ← SRC[127];
r64[63:16] ← ZERO_FILL;
```

Intel C/C++ Compiler Intrinsic Equivalent

PMOVMSKB int _mm_movemask_epi8 (__m128i a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

PMOVSX - Packed Move with Sign Extend

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 20 /r PMOVSXBW xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1
66 0f 38 21 /r PMOVSXBD xmm1, xmm2/m32	A	V/V	SSE4_1	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1
66 0f 38 22 /r PMOVSXBQ xmm1, xmm2/m16	A	V/V	SSE4_1	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1
66 0f 38 23/r PMOVSXWD xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1
66 0f 38 24 /r PMOVSXWQ xmm1, xmm2/m32	A	V/V	SSE4_1	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1
66 0f 38 25 /r PMOVSXDQ xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1
VEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1, xmm2/m64	A	V/V	AVX	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1
VEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1, xmm2/m32	A	V/V	AVX	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1, xmm2/m16	A	V/V	AVX	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1
VEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1, xmm2/m64	A	V/V	AVX	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1
VEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1, xmm2/m32	A	V/V	AVX	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1
VEX.128.66.0F38.WIG 25 /r VPMOVSXDQ xmm1, xmm2/m64	A	V/V	AVX	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Packed byte, word, or dword integers in the low bytes of the source operand (second operand) are sign extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
Packed_Sign_Extend_BYTE_to_WORD
DEST[15:0] ← SignExtend(SRC[7:0]);
```

```
DEST[31:16] ← SignExtend(SRC[15:8]);
DEST[47:32] ← SignExtend(SRC[23:16]);
DEST[63:48] ← SignExtend(SRC[31:24]);
DEST[79:64] ← SignExtend(SRC[39:32]);
DEST[95:80] ← SignExtend(SRC[47:40]);
DEST[111:96] ← SignExtend(SRC[55:48]);
DEST[127:112] ← SignExtend(SRC[63:56]);

Packed_Sign_Extend_BYTE_to_DWORD
DEST[31:0] ← SignExtend(SRC[7:0]);
DEST[63:32] ← SignExtend(SRC[15:8]);
DEST[95:64] ← SignExtend(SRC[23:16]);
DEST[127:96] ← SignExtend(SRC[31:24]);

Packed_Sign_Extend_BYTE_to_QWORD
DEST[63:0] ← SignExtend(SRC[7:0]);
DEST[127:64] ← SignExtend(SRC[15:8]);

Packed_Sign_Extend_WORD_to_DWORD
DEST[31:0] ← SignExtend(SRC[15:0]);
DEST[63:32] ← SignExtend(SRC[31:16]);
DEST[95:64] ← SignExtend(SRC[47:32]);
DEST[127:96] ← SignExtend(SRC[63:48]);

Packed_Sign_Extend_WORD_to_QWORD
DEST[63:0] ← SignExtend(SRC[15:0]);
DEST[127:64] ← SignExtend(SRC[31:16]);

Packed_Sign_Extend_DWORD_to_QWORD
DEST[63:0] ← SignExtend(SRC[31:0]);
DEST[127:64] ← SignExtend(SRC[63:32]);
```

VPMOVSXBW

```
Packed_Sign_Extend_BYTE_to_WORD()
DEST[255:128] ← 0
```

VPMOVSXBD

```
Packed_Sign_Extend_BYTE_to_DWORD()
DEST[255:128] ← 0
```

VPMOVSXBQ

```
Packed_Sign_Extend_BYTE_to_QWORD()
DEST[255:128] ← 0
```

VPMOVSXWD
Packed_Sign_Extend_WORD_to_DWORD()
DEST[255:128] ← 0

VPMOVSXWQ
Packed_Sign_Extend_WORD_to_QWORD()
DEST[255:128] ← 0

VPMOVSXDQ
Packed_Sign_Extend_DWORD_to_QWORD()
DEST[255:128] ← 0

PMOVSXBW
Packed_Sign_Extend_BYTE_to_WORD()
DEST[255:128] (Unmodified)

PMOVSXBD
Packed_Sign_Extend_BYTE_to_DWORD()
DEST[255:128] (Unmodified)

PMOVSXBQ
Packed_Sign_Extend_BYTE_to_QWORD()
DEST[255:128] (Unmodified)

PMOVSXWD
Packed_Sign_Extend_WORD_to_DWORD()
DEST[255:128] (Unmodified)

PMOVSXWQ
Packed_Sign_Extend_WORD_to_QWORD()
DEST[255:128] (Unmodified)

PMOVSXDQ
Packed_Sign_Extend_DWORD_to_QWORD()
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMOVSXBW __m128i _mm_ cvtepi8_epi16 (__m128i a);

PMOVSXBD __m128i _mm_ cvtepi8_epi32 (__m128i a);

PMOVSXBQ __m128i _mm_ cvtepi8_epi64 (__m128i a);

PMOVSXWD __m128i _mm_ cvtepi16_epi32 (__m128i a);

PMOVSXWQ __m128i _mm_ cvtepi16_epi64 (__m128i a);

PMOVSXDQ __m128i _mm_ cvtepi32_epi64 (__m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

PMOVZX - Packed Move with Zero Extend

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 30 /r PMOVZXBW xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1
66 0f 38 31 /r PMOVZXBD xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1
66 0f 38 32 /r PMOVZXBQ xmm1, xmm2/m16	A	V/V	SSE4_1	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1
66 0f 38 33 /r PMOVZXWD xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1
66 0f 38 34 /r PMOVZXWQ xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1
66 0f 38 35 /r PMOVZXDQ xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1
VEX.128.66.0F38.WIG 30 /r VPMOVZXBW xmm1, xmm2/m64	A	V/V	AVX	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1
VEX.128.66.0F38.WIG 31 /r VPMOVZXBD xmm1, xmm2/m32	A	V/V	AVX	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.WIG 32 /r VPMOVZXBQ xmm1, xmm2/m16	A	V/V	AVX	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1
VEX.128.66.0F38.WIG 33 /r VPMOVZXWD xmm1, xmm2/m64	A	V/V	AVX	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1
VEX.128.66.0F38.WIG 34 /r VPMOVZXWQ xmm1, xmm2/m32	A	V/V	AVX	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1
VEX.128.66.0F38.WIG 35 /r VPMOVZXDQ xmm1, xmm2/m64	A	V/V	AVX	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Packed byte, word, or dword integers in the low bytes of the source operand (second operand) are zero extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
Packed_Zero_Extend_BYTE_to_WORD
DEST[15:0] ← ZeroExtend(SRC[7:0]);
DEST[31:16] ← ZeroExtend(SRC[15:8]);
```

DEST[47:32] ← ZeroExtend(SRC[23:16]);
DEST[63:48] ← ZeroExtend(SRC[31:24]);
DEST[79:64] ← ZeroExtend(SRC[39:32]);
DEST[95:80] ← ZeroExtend(SRC[47:40]);
DEST[111:96] ← ZeroExtend(SRC[55:48]);
DEST[127:112] ← ZeroExtend(SRC[63:56]);

Packed_Zero_Extend_BYTE_to_DWORD
DEST[31:0] ← ZeroExtend(SRC[7:0]);
DEST[63:32] ← ZeroExtend(SRC[15:8]);
DEST[95:64] ← ZeroExtend(SRC[23:16]);
DEST[127:96] ← ZeroExtend(SRC[31:24]);

Packed_Zero_Extend_BYTE_to_QWORD
DEST[63:0] ← ZeroExtend(SRC[7:0]);
DEST[127:64] ← ZeroExtend(SRC[15:8]);

Packed_Zero_Extend_WORD_to_DWORD
DEST[31:0] ← ZeroExtend(SRC[15:0]);
DEST[63:32] ← ZeroExtend(SRC[31:16]);
DEST[95:64] ← ZeroExtend(SRC[47:32]);
DEST[127:96] ← ZeroExtend(SRC[63:48]);

Packed_Zero_Extend_WORD_to_QWORD
DEST[63:0] ← ZeroExtend(SRC[15:0]);
DEST[127:64] ← ZeroExtend(SRC[31:16]);

Packed_Zero_Extend_DWORD_to_QWORD
DEST[63:0] ← ZeroExtend(SRC[31:0]);
DEST[127:64] ← ZeroExtend(SRC[63:32]);

VPMOVZXBW
Packed_Zero_Extend_BYTE_to_WORD()
DEST[255:128] ← 0

VPMOVZXBD
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[255:128] ← 0

VPMOVZXBQ
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[255:128] ← 0

VPMOVZXWD
Packed_Zero_Extend_WORD_to_DWORD()
DEST[255:128] ← 0

VPMOVZXWQ
Packed_Zero_Extend_WORD_to_QWORD()
DEST[255:128] ← 0

VPMOVZXDQ
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[255:128] ← 0

PMOVZXBW
Packed_Zero_Extend_BYTE_to_WORD()
DEST[255:128] (Unmodified)

PMOVZXBD
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[255:128] (Unmodified)

PMOVZXBQ
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[255:128] (Unmodified)

PMOVZXWD
Packed_Zero_Extend_WORD_to_DWORD()
DEST[255:128] (Unmodified)

PMOVZXWQ
Packed_Zero_Extend_WORD_to_QWORD()
DEST[255:128] (Unmodified)

PMOVZXDQ
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMOVZXBW __m128i _mm_ cvtepu8_epi16 (__m128i a);

PMOVZXBD __m128i _mm_ cvtepu8_epi32 (__m128i a);

PMOVZXBQ __m128i _mm_ cvtepu8_epi64 (__m128i a);

PMOVZXWD __m128i _mm_ cvtepu16_epi32 (__m128i a);

PMOVZXWQ __m128i _mm_ cvtepu16_epi64 (__m128i a);

PMOVZXDQ __m128i _mm_ cvtepu32_epi64 (__m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

PMULHUW - Multiply Packed Unsigned Integers and Store High Result

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E4 /r PMULHUW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed unsigned word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.
VEX.NDS.128.66.0F.WIG E4 /r VPMULHUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD unsigned multiply of the packed unsigned word integers in the first source operand and the second source operand, and stores the high 16 bits of each 32-bit intermediate results in the destination operand.

The second source operand is an XMM register or a 128-bit memory location. The destination operand and first source operands are XMM registers.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMULHUW (VEX.128 encoded version)

TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]

TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[255:128] ← 0

PMULHUW (128-bit Legacy SSE version)
TEMP0[31:0] ← DEST[15:0] * SRC[15:0]
TEMP1[31:0] ← DEST[31:16] * SRC[31:16]
TEMP2[31:0] ← DEST[47:32] * SRC[47:32]
TEMP3[31:0] ← DEST[63:48] * SRC[63:48]
TEMP4[31:0] ← DEST[79:64] * SRC[79:64]
TEMP5[31:0] ← DEST[95:80] * SRC[95:80]
TEMP6[31:0] ← DEST[111:96] * SRC[111:96]
TEMP7[31:0] ← DEST[127:112] * SRC[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMULHUW __m128i _mm_mulhi_epu16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMULHRSW - Multiply Packed Unsigned Integers with Round and Shift

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 0B /r PMULHRSW xmm1, xmm2/m128	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1.
VEX.NDS.128.66.0F38.WIG 0B /r VPMULHRSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PMULHRSW multiplies vertically each signed 16-bit integer from the first source operand with the corresponding signed 16-bit integer of the second source operand, producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand. The first source and destination operands are XMM registers. The second source operand is an XMM register or 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPMULHRSW (VEX.128 encoded version)

```
temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
```

temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[255:128] ← 0

PMULHRSW (128-bit Legacy SSE version)
temp0[31:0] ← INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((DEST[79:64] * SRC[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((DEST[95:80] * SRC[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((DEST[111:96] * SRC[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((DEST[127:112] * SRC[127:112) >>14) + 1
DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMULHRSW __m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMULHW - Multiply Packed Integers and Store High Result

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E5 /r PMULHW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.
VEX.NDS.128.66.0F.WIG E5 /r VPMULHW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD signed multiply of the packed signed word integers in the first source operand and the second source operand, and stores the high 16 bits of each intermediate 32-bit result in the destination operand. The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMULHW (VEX.128 encoded version)

TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]

TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[255:128] ← 0

PMULHW (128-bit Legacy SSE version)
TEMP0[31:0] ← DEST[15:0] * SRC[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16]
TEMP2[31:0] ← DEST[47:32] * SRC[47:32]
TEMP3[31:0] ← DEST[63:48] * SRC[63:48]
TEMP4[31:0] ← DEST[79:64] * SRC[79:64]
TEMP5[31:0] ← DEST[95:80] * SRC[95:80]
TEMP6[31:0] ← DEST[111:96] * SRC[111:96]
TEMP7[31:0] ← DEST[127:112] * SRC[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMULHW __m128i _mm_mulhi_epi16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMULLW/PMULLD - Multiply Packed Integers and Store Low Result

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F D5 /r PMULLW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the low 16 bits of the results in xmm1.
66 0F 38 40 /r PMULLD xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply the packed dword signed integers in xmm1 and xmm2/m128 and store the low 32 bits of each product in xmm1
VEX.NDS.128.66.0F.WIG D5 /r VPMULLW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the low 16 bits of the results in xmm1.
VEX.NDS.128.66.0F38.WIG 40 /r VPMULLD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD signed multiply of the packed signed word (dword) integers in the first source operand and the second source operand and stores the low 16(32) bits of each intermediate 32-bit(64-bit) result in the destination operand. (Figure 4-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B,shows this operation when using 64-bit operands.) The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPMULLD (VEX.128 encoded version)

Temp0[63:0] ← SRC1[31:0] * SRC2[31:0]
Temp1[63:0] ← SRC1[63:32] * SRC2[63:32]
Temp2[63:0] ← SRC1[95:64] * SRC2[95:64]
Temp3[63:0] ← SRC1[127:96] * SRC2[127:96]
DEST[31:0] ← Temp0[31:0]
DEST[63:32] ← Temp1[31:0]
DEST[95:64] ← Temp2[31:0]
DEST[127:96] ← Temp3[31:0]
DEST[255:128] ← 0

PMULLD (128-bit Legacy SSE version)

Temp0[63:0] ← DEST[31:0] * SRC[31:0]
Temp1[63:0] ← DEST[63:32] * SRC[63:32]
Temp2[63:0] ← DEST[95:64] * SRC[95:64]
Temp3[63:0] ← DEST[127:96] * SRC[127:96]
DEST[31:0] ← Temp0[31:0]
DEST[63:32] ← Temp1[31:0]
DEST[95:64] ← Temp2[31:0]
DEST[127:96] ← Temp3[31:0]
DEST[255:128] (Unmodified)

VPMULLW (VEX.128 encoded version)

Temp0[31:0] ← SRC1[15:0] * SRC2[15:0]
Temp1[31:0] ← SRC1[31:16] * SRC2[31:16]
Temp2[31:0] ← SRC1[47:32] * SRC2[47:32]
Temp3[31:0] ← SRC1[63:48] * SRC2[63:48]
Temp4[31:0] ← SRC1[79:64] * SRC2[79:64]
Temp5[31:0] ← SRC1[95:80] * SRC2[95:80]
Temp6[31:0] ← SRC1[111:96] * SRC2[111:96]
Temp7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← Temp0[15:0]
DEST[31:16] ← Temp1[15:0]
DEST[47:32] ← Temp2[15:0]
DEST[63:48] ← Temp3[15:0]
DEST[79:64] ← Temp4[15:0]
DEST[95:80] ← Temp5[15:0]
DEST[111:96] ← Temp6[15:0]
DEST[127:112] ← Temp7[15:0]

DEST[255:128] ← 0

PMULLW (128-bit Legacy SSE version)
Temp0[31:0] ← DEST[15:0] * SRC[15:0]
Temp1[31:0] ← DEST[31:16] * SRC[31:16]
Temp2[31:0] ← DEST[47:32] * SRC[47:32]
Temp3[31:0] ← DEST[63:48] * SRC[63:48]
Temp4[31:0] ← DEST[79:64] * SRC[79:64]
Temp5[31:0] ← DEST[95:80] * SRC[95:80]
Temp6[31:0] ← DEST[111:96] * SRC[111:96]
Temp7[31:0] ← DEST[127:112] * SRC[127:112]
DEST[15:0] ← Temp0[15:0]
DEST[31:16] ← Temp1[15:0]
DEST[47:32] ← Temp2[15:0]
DEST[63:48] ← Temp3[15:0]
DEST[79:64] ← Temp4[15:0]
DEST[95:80] ← Temp5[15:0]
DEST[111:96] ← Temp6[15:0]
DEST[127:112] ← Temp7[15:0]
DEST[127:96] ← Temp3[31:0];
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMULLW __m128i _mm_mullo_epi16 (__m128i a, __m128i b)

PMULLUD __m128i _mm_mullo_epi32(__m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMULUDQ - Multiply Packed Unsigned Doubleword Integers

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F F4 /r PMULUDQ xmm1, xmm2/m128	A	V/V	SSE2	Multiply packed unsigned doubleword integers in xmm1 by packed unsigned doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.NDS.128.66.0F.WIG F4 /r VPMULUDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128, and store the quadword results in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the first source operand by the second source operand and stores the result in the destination operand. The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or an 128-bit memory location. The first source operand is two packed doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPMULUDQ (VEX.128 encoded version)

DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:64] ← SRC1[95:64] * SRC2[95:64]
DEST[255:128] ← 0

PMULUDQ (128-bit Legacy SSE version)

DEST[63:0] ← DEST[31:0] * SRC[31:0]
DEST[127:64] ← DEST[95:64] * SRC[95:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMULUDQ __m128i _mm_mul_epu32 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMULDQ - Multiply Packed Doubleword Integers

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 28 /r PMULDQ xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply packed signed doubleword integers in xmm1 by packed signed doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.NDS.128.66.0F38.WIG 28 /r VPMULDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128, and store the quadword results in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the first source operand by the second source operand and stores the result in the destination operand. The second source operand is two packed signed doubleword integers stored in the first (low) and third doublewords of an XMM register or an 128-bit memory location. The first source operand is two packed signed doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed signed quadword integers stored in an XMM register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPMULDQ (VEX.128 encoded version)

DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:64] ← SRC1[95:64] * SRC2[95:64]
DEST[255:128] ← 0

PMULDQ (128-bit Legacy SSE version)

DEST[63:0] ← DEST[31:0] * SRC[31:0]
DEST[127:64] ← DEST[95:64] * SRC[95:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMULDQ __m128i _mm_mul_epi32(__m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

POR - Bitwise Logical Or

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F EB /r POR xmm1, xmm2/m128	A	V/V	SSE2	Bitwise OR of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG EB /r VPOR xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise OR of xmm2/m128 and xmm3.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical OR operation on the second source operand and the first source operand and stores the result in the destination operand. The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPOR (VEX.128 encoded version)

DEST ← SRC1 OR SRC2
DEST[255:128] ← 0

POR (128-bit Legacy SSE version)

DEST ← DEST OR SRC
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

POR __m128i _mm_or_si128 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PSADBW - Compute Sum of Absolute Differences

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F F6 /r PSADBW xmm1, xmm2/m128	A	V/V	SSE2	Computes the absolute differences of the packed unsigned byte integers from xmm2 /m128 and xmm1; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.NDS.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Computes the absolute differences of the packed unsigned byte integers from xmm3 /m128 and xmm2; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the absolute value of the difference of packed groups of 8 unsigned byte integers from the second operand and from the first source operand. The first 8 differences are summed to produce an unsigned word integer that is stored in the low word of the destination; the second 8 differences are summed to produce an unsigned word in bit 79:64 of the destination. The remaining words of the destination are set to 0.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPSADBW (VEX.128 encoded version)

```
TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 ← ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000
DEST[255:128] ← 0
```

PSADBW (128-bit Legacy SSE version)

```
TEMP0 ← ABS(DEST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 ← ABS(DEST[127:120] - SRC[127:120])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000

DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

PSADBW __m128i _mm_sad_epu8(__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PSHUFB - Packed Shuffle Bytes

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 00 /r PSHUFB xmm1, xmm2/m128	A	V/V	SSSE3	Shuffle bytes in xmm1 according to contents of xmm2/m128.
VEX.NDS.128.66.0F38.WIG 00 /r VPSHUFB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Shuffle bytes in xmm2 according to contents of xmm3/m128.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shuffles bytes in the first source operand according to the shuffle control mask in the second source operand. The instruction permutes byte data in the first source operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte element in the shuffle control mask provides an index field to select the byte element in the first source operand. The index field is defined as the least significant 4 bits of each byte element of the shuffle control mask. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is the first operand, the first source operand is the second operand, the second source operand is the third operand. Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPSHUFB (VEX.128 encoded version)

```
for i = 0 to 15 {
    if (SRC2[(i * 8)+7] == 1 ) then
```

```
        DEST[(i*8)+7..(i*8)+0] ← 0;
        else
        index[3..0] ← SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← SRC1[(index*8+7)..(index*8+0)];
    endif
}
DEST[255:128] ← 0
```

PSHUFB (128-bit Legacy SSE version)

```
for i = 0 to 15 {
    if (SRC[(i * 8)+7] == 1 ) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
        else
        index[3..0] ← SRC[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← DEST[(index*8+7)..(index*8+0)];
    endif
}
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFB __m128i _mm_shuffle_epi8(__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PSHUFD - Shuffle Packed Doublewords

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 70 /r ib PSHUFD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.128.66.0F.WIG 70 /r ib VPSHUFD xmm1, xmm2/m128, imm8	A	V/V	AVX	Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Copies doublewords from source operand and inserts them in the destination operand at the locations selected with the immediate control operand. Figure 5-23 shows the operation of the PSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location in the destination operand. For example, bits 0 and 1 of the order operand select the contents of doubleword 0 of the destination operand. The encoding of bits 0 and 1 of the order operand (see the field encoding in Figure 5-23) determines which doubleword from the source operand will be copied to doubleword 0 of the destination operand.

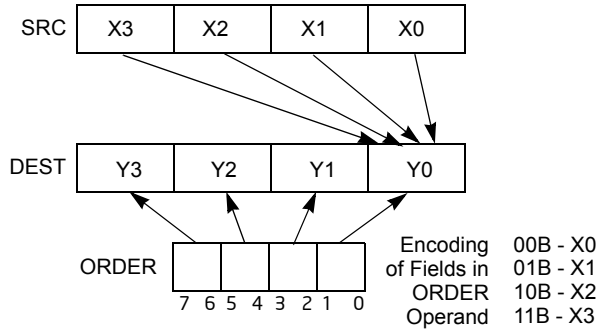


Figure 5-23. PSHUFD Instruction Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPSHUFD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[255:128] ← 0
```

PSHUFD (128-bit Legacy SSE version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFD __m128i _mm_shuffle_epi32(__m128i a, int n)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

PSHUFHW - Shuffle Packed High Words

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 70 /r ib PSHUFHW xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.128.F3.0F.WIG 70 /r ib VPSHUFHW xmm1, xmm2/m128, imm8	A	V/V	AVX	Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Copies words from the high quadword of the source operand and inserts them in the high quadword of the destination operand at word locations selected with the immediate operand. This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-7 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B. For the PSHUFHW instruction, each 2-bit field in the immediate operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the source operand is copied to the low quadword of the destination operand.

The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Note that this instruction permits a word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPSHUFHW (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[255:128] ← 0
```

PSHUFHW (128-bit Legacy SSE version)

```
DEST[63:0] ← SRC[63:0]
DEST[79:64] ← (SRC >> (imm[1:0] *16))[79:64]
DEST[95:80] ← (SRC >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC >> (imm[7:6] * 16))[79:64]
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFHW __m128i _mm_shufflehi_epi16(__m128i a, int n)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

PSHUFLW - Shuffle Packed Low Words

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 70 /r ib PSHUFLW xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.128.F2.0F.WIG 70 /r ib VPSHUFLW xmm1, xmm2/m128, imm8	A	V/V	AVX	Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Copies words from the low quadword of the source operand and inserts them in the low quadword of the destination operand at word locations selected with the immediate operand. This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 5-23. For the PSHUFLW instruction, each 2-bit field in the immediate operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3) from the low quadword of the source operand to be copied to the destination operand. The high quadword of the source operand is copied to the high quadword of the destination operand.

The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Note that this instruction permits a word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise instructions will #UD.

Operation

VPSHUFLW (VEX.128 encoded version)

DEST[15:0] ← (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC[127:64]
DEST[255:128] ← 0

PSHUFLW (128-bit Legacy SSE version)
DEST[15:0] ← (SRC >> (imm[1:0] *16))[15:0]
DEST[31:16] ← (SRC >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC[127:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFLW __m128i _mm_shufflelo_epi16(__m128i a, int n)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

PSIGNB/PSIGNW/PSIGND - Packed SIGN

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 08 /r PSIGNB xmm1, xmm2/m128	A	V/V	SSSE3	Negate/zero/preserve packed byte integers in xmm1 depending on the corresponding sign in xmm2/m128.
66 0F 38 09 /r PSIGNW xmm1, xmm2/m128	A	V/V	SSSE3	Negate/zero/preserve packed word integers in xmm1 depending on the corresponding sign in xmm2/m128.
66 0F 38 0A /r PSIGND xmm1, xmm2/m128	A	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in xmm1 depending on the corresponding sign in xmm2/m128.
VEX.NDS.128.66.0F38.WIG 08 /r VPSIGNB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Negate/zero/preserve packed byte integers in xmm2 depending on the corresponding sign in xmm3/m128.
VEX.NDS.128.66.0F38.WIG 09 /r VPSIGNW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Negate/zero/preserve packed word integers in xmm2 depending on the corresponding sign in xmm3/m128.
VEX.NDS.128.66.0F38.WIG 0A /r VPSIGND xmm1, xmm2, xmm3/m128	B	V/V	AVX	Negate/zero/preserve packed doubleword integers in xmm2 depending on the corresponding sign in xmm3/m128.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PSIGNB/PSIGNW/PSIGND negates each data element of the first source operand if the signed integer value of the corresponding data element in the second source operand is less than zero. If the signed integer value of a data element in the second source operand is positive, the corresponding data element in the first source operand is unchanged. If a data element in the second source operand is zero, the corresponding data element in the first source operand is set to zero.

PSIGNB operates on signed bytes. PSIGNW operates on 16-bit signed words. PSIGND operates on signed 32-bit integers.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

```
BYTE_SIGN(SRC1, SRC2)
    if (SRC2[7..0] < 0 )
        DEST[7...0] ← Neg(SRC1[7...0])
    else if(SRC2[7..0] == 0 )
        DEST[7...0] ← 0
    else if(SRC2[7..0] > 0 )
        DEST[7...0] ← SRC1[7...0]
Repeat operation for 2nd through 15th bytes
    if (SRC2[127..120] < 0 )
        DEST[127...120] ← Neg(SRC1[127...120])
    else if(SRC2[127.. 120] == 0 )
        DEST[127...120] ← 0
    else if(SRC2[127.. 120] > 0 )
        DEST[127...120] ← SRC1[127...120]

WORD_SIGN(SRC1, SRC2)
    if (SRC2[15..0] < 0 )
        DEST[15...0] ← Neg(SRC1[15...0])
    else if(SRC2[15..0] == 0 )
        DEST[15...0] ← 0
    else if(SRC2[15..0] > 0 )
        DEST[15...0] ← SRC1[15...0]
```

```
Repeat operation for 2nd through 7th words
    if (SRC2[127..112] < 0 )
        DEST[127...112] ← Neg(SRC1[127...112])
    else if(SRC2[127.. 112] == 0 )
        DEST[127...112] ← 0
    else if(SRC2[127.. 112] > 0 )
        DEST[127...112] ← SRC1[127...112]

DWORD_SIGN(SRC1, SRC2)
    if (SRC2[31..0] < 0 )
        DEST[31...0] ← Neg(SRC1[31...0])
    else if(SRC2[31..0] == 0 )
        DEST[31...0] ← 0
    else if(SRC2[31..0] > 0 )
        DEST[31...0] ← SRC1[31...0]
Repeat operation for 2nd through 3rd double words
    if (SRC2[127..96] < 0 )
        DEST[127...96] ← Neg(SRC1[127...96])
    else if(SRC2[127.. 96] == 0 )
        DEST[127...96] ←0
    else if(SRC2[127.. 96] > 0 )
        DEST[127...96] ← SRC1[127...96]
```

VPSIGNB (VEX.128 encoded version)

```
DEST[127:0] ←BYTE_SIGN(SRC1, SRC2)
DEST[255:128] ← 0
```

PSIGNB (128-bit Legacy SSE version)

```
DEST[127:0] ←BYTE_SIGN(DEST, SRC)
DEST[255:128] (Unmodified)
```

VPSIGNW (VEX.128 encoded version)

```
DEST[127:0] ←WORD_SIGN(SRC1, SRC2)
DEST[255:128] ← 0
```

PSIGNW (128-bit Legacy SSE version)

```
DEST[127:0] ←WORD_SIGN(DEST, SRC)
DEST[255:128] (Unmodified)
```

VPSIGND (VEX.128 encoded version)

```
DEST[127:0] ←DWORD_SIGN(SRC1, SRC2)
DEST[255:128] ← 0
```

PSIGND (128-bit Legacy SSE version)
DEST[127:0] ←DWORD_SIGN(DEST, SRC)
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PSIGNB __m128i _mm_sign_epi8 (__m128i a, __m128i b)

PSIGNW __m128i _mm_sign_epi16 (__m128i a, __m128i b)

PSIGND __m128i _mm_sign_epi32 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PSLLDQ - Byte Shift Left

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /7 ib PSLLDQ xmm1, imm8	A	V/V	SSE2	Shift xmm1 left by imm8 bytes while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /7 ib VPSLLDQ xmm1, xmm2, imm8	B	V/V	AVX	Shift xmm2 left by imm8 bytes while shifting in 0s and store result in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (r, w)	NA	NA	NA
B	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Shifts the source operand to the left by the number of bytes specified in the count operand. The empty low-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s.

The source and destination operands are XMM registers. The count operand is an 8-bit immediate.

128-bit Legacy SSE version: The source and destination operands are the same. Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

Operation

VPSLLDQ (VEX.128 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← SRC << (TEMP * 8)
DEST[255:128] ← 0
```

PSLLDQ(128-bit Legacy SSE version)

```
TEMP ← COUNT
```

```
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST << (TEMP * 8)
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

PSLLDQ __m128i _mm_slli_si128 (__m128i a, int imm)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.L = 1.

PSRLDQ - Byte Shift Right

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /3 ib PSRLDQ xmm1, imm8	A	V/V	SSE2	Shift xmm1 right by imm8 bytes while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /3 ib VPSRLDQ xmm1, xmm2, imm8	B	V/V	AVX	Shift xmm2 right by imm8 bytes while shifting in 0s.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (r, w)	NA	NA	NA
B	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Shifts the source operand to the right by the number of bytes specified in the count operand. The empty high-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s.

The source and destination operands are XMM registers. The count operand is an 8-bit immediate.

128-bit Legacy SSE version: The source and destination operands are the same. Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

Operation

VPSRLDQ (VEX.128 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← SRC >> (TEMP * 8)
DEST[255:128] ← 0
```

PSRLDQ(128-bit Legacy SSE version)

```
TEMP ← COUNT
```

```
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST >> (TEMP * 8)
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

PSRLDQ __m128i _mm_srli_si128 (__m128i a, int imm)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.L = 1.

PSLLW/PSLLD/PSLLQ - Bit Shift Left

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F F1/r PSLLW xmm1, xmm2/m128	A	V/V	SSE2	Shift words in xmm1 left by amount specified in xmm2/m128 while shifting in 0s.
66 0F 71 /6 ib PSLLW xmm1, imm8	B	V/V	SSE2	Shift words in xmm1 left by imm8 while shifting in 0s.
66 0F F2 /r PSLLD xmm1, xmm2/m128	A	V/V	SSE2	Shift doublewords in xmm1 left by amount specified in xmm2/m128 while shifting in 0s.
66 0F 72 /6 ib PSLLD xmm1, imm8	B	V/V	SSE2	Shift doublewords in xmm1 left by imm8 while shifting in 0s.
66 0F F3 /r PSLLQ xmm1, xmm2/m128	A	V/V	SSE2	Shift quadwords in xmm1 left by amount specified in xmm2/m128 while shifting in 0s.
66 0F 73 /6 ib PSLLQ xmm1, imm8	B	V/V	SSE2	Shift quadwords in xmm1 left by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG F1 /r VPSLLW xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW xmm1, xmm2, imm8	D	V/V	AVX	Shift words in xmm2 left by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG F2 /r VPSLLD xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDD.128.66.0F.WIG 72 /6 ib VPSLLD xmm1, xmm2, imm8	D	V/V	AVX	Shift doublewords in xmm2 left by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG F3 /r VPSLLQ xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /6 ib VPSLLQ xmm1, xmm2, imm8	D	V/V	AVX	Shift quadwords in xmm2 left by imm8 while shifting in 0s.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (r, w)	NA	NA	NA
C	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
D	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s.

The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.

The PSLLW instruction shifts each of the words in the first source operand to the left by the number of bits specified in the count operand; the PSLLD instruction shifts each of the doublewords in the first source operand; and the PSLLQ instruction shifts the quadword (or quadwords) in the first source operand.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. For shifts with an immediate count (VEX.128.66.0F 71-73 /6), VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

Operation

```
LOGICAL_LEFT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[15:0] ← ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ← ZeroExtend(SRC[127:112] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ← ZeroExtend(SRC[127:96] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] << COUNT);
FI;

VPSLLW (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(SRC1, SRC2)
```

DEST[255:128] ← 0

VPSLLW (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)
DEST[255:128] ← 0

PSLLW (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
DEST[255:128] (Unmodified)

PSLLW (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
DEST[255:128] (Unmodified)

VPSLLD (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
DEST[255:128] ← 0

VPSLLD (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[255:128] ← 0

PSLLD (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
DEST[255:128] (Unmodified)

PSLLD (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
DEST[255:128] (Unmodified)

VPSLLQ (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)
DEST[255:128] ← 0

VPSLLQ (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)
DEST[255:128] ← 0

PSLLQ (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)
DEST[255:128] (Unmodified)

PSLLQ (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(DEST, imm8)
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PSLLW __m128i _mm_slli_epi16 (__m128i m, int count)

PSLLW __m128i _mm_sll_epi16 (__m128i m, __m128i count)

PSLLD __m128i _mm_slli_epi32 (__m128i m, int count)

PSLLD __m128i _mm_sll_epi32 (__m128i m, __m128i count)

PSLLQ __m128i _mm_slli_epi64 (__m128i m, int count)

PSLLQ __m128i _mm_sll_epi64 (__m128i m, __m128i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4 and 7 for non-VEX-encoded instructions.

#UD	If VEX.L = 1.

PSRAW/PSRAD - Bit Shift Arithmetic Right

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E1/r PSRAW xmm1, xmm2/m128	A	V/V	SSE2	Shift words in xmm1 right by amount specified in xmm2/m128 while shifting in sign bits.
66 0F 71 /4 ib PSRAW xmm1, imm8	B	V/V	SSE2	Shift words in xmm1 right by imm8 while shifting in sign bits.
66 0F E2 /r PSRAD xmm1, xmm2/m128	A	V/V	SSE2	Shift doublewords in xmm1 right by amount specified in xmm2/m128 while shifting in sign bits.
66 0F 72 /4 ib PSRAD xmm1, imm8	B	V/V	SSE2	Shift doublewords in xmm1 right by imm8 while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E1 /r VPSRAW xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 71 /4 ib VPSRAW xmm1, xmm2, imm8	D	V/V	AVX	Shift words in xmm2 right by imm8 while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E2 /r VPSRAD xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 72 /4 ib VPSRAD xmm1, xmm2, imm8	D	V/V	AVX	Shift doublewords in xmm2 right by imm8 while shifting in sign bits.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (r, w)	NA	NA	NA
C	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
D	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count operand. As the bits in the data elements are shifted left, the empty high-order bits are filled with the initial value of the sign bit of the data. If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is filled with the initial value of the sign bit.

The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.

The PSRAW instruction shifts each of the words in the first source operand to the right by the number of bits specified in the count operand; the PSRAD instruction shifts each of the doublewords in the first source operand; and the PSRAQ instruction shifts the quadword (or quadwords) in the first source operand.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged. : Bits (255:128) of the corresponding YMM destination register remain unchanged. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. For shifts with an immediate count (VEX.128.66.0F 71-73 /4), VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD. : Bits (255:128) of the corresponding YMM destination register remain unchanged. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

Operation

```
ARITHMETIC_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    COUNT ← 16
FI
DEST[15:0] ← SignExtend(SRC[15:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
```

```
DEST[127:112] ← SignExtend(SRC[127:112] >> COUNT);
```

ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)

```
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    COUNT ← 32
FI
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
(* Repeat shift operation for 2nd through 3rd words *)
DEST[127:96] ← SignExtend(SRC[127:96] >> COUNT);
```

VPSRAW (xmm, xmm, xmm/m128)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[255:128] ← 0
```

VPSRAW (xmm, imm8)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[255:128] ← 0
```

PSRAW (xmm, xmm, xmm/m128)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[255:128] (Unmodified)
```

PSRAW (xmm, imm8)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[255:128] (Unmodified)
```

VPSRAD (xmm, xmm, xmm/m128)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[255:128] ← 0
```

VPSRAD (xmm, imm8)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[255:128] ← 0
```

PSRAD (xmm, xmm, xmm/m128)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[255:128] (Unmodified)
```

PSRAD (xmm, imm8)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

PSRAW __m128i _mm_srai_epi16 (__m128i m, int count)

PSRAW __m128i _mm_sra_epi16 (__m128i m, __m128i count)

PSRAD __m128i _mm_srai_epi32 (__m128i m, int count)

PSRAD __m128i _mm_sra_epi32 (__m128i m, __m128i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4 and 7 for non-VEX-encoded instructions.

#UD	If VEX.L = 1.

PSRLW/PSRLD/PSRLQ - Shift Packed Data Right Logical

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F D1 /r PSRLW xmm1, xmm2/m128	A	V/V	SSE2	Shift words in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.
66 0F 71 /2 ib PSRLW xmm1, imm8	B	V/V	SSE2	Shift words in xmm1 right by imm8 while shifting in 0s.
66 0F D2 /r PSRLD xmm1, xmm2/m128	A	V/V	SSE2	Shift doublewords in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.
66 0F 72 /2 ib PSRLD xmm1, imm8	B	V/V	SSE2	Shift doublewords in xmm1 right by imm8 while shifting in 0s.
66 0F D3 /r PSRLQ xmm1, xmm2/m128	A	V/V	SSE2	Shift quadwords in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.
66 0F 73 /2 ib PSRLQ xmm1, imm8	B	V/V	SSE2	Shift quadwords in xmm1 right by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG D1 /r VPSRLW xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /2 ib VPSRLW xmm1, xmm2, imm8	D	V/V	AVX	Shift words in xmm2 right by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG D2 /r VPSRLD xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDD.128.66.0F.WIG 72 /2 ib VPSRLD xmm1, xmm2, imm8	D	V/V	AVX	Shift doublewords in xmm2 right by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG D3 /r VPSRLQ xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /2 ib VPSRLQ xmm1, xmm2, imm8	D	V/V	AVX	Shift quadwords in xmm2 right by imm8 while shifting in 0s.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (r, w)	NA	NA	NA
C	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
D	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s.

The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.

The PSRLW instruction shifts each of the words in the first source operand to the right by the number of bits specified in the count operand; the PSRLD instruction shifts each of the doublewords in the first source operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the first source operand.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. For shifts with an immediate count (VEX.128.66.0F 71-73 /2), VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

Operation

```
LOGICAL_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[15:0] ← ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ← ZeroExtend(SRC[127:112] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ← ZeroExtend(SRC[127:96] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] >> COUNT);
FI;

VPSRLW (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)
```

DEST[255:128] ← 0

VPSRLW (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[255:128] ← 0

PSRLW (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[255:128] (Unmodified)

PSRLW (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[255:128] (Unmodified)

VPSRLD (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[255:128] ← 0

VPSRLD (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[255:128] ← 0

PSRLD (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[255:128] (Unmodified)

PSRLD (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[255:128] (Unmodified)

VPSRLQ (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, SRC2)
DEST[255:128] ← 0

VPSRLQ (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, imm8)
DEST[255:128] ← 0

PSRLQ (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(DEST, SRC)
DEST[255:128] (Unmodified)

PSRLQ (xmm, imm8)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(DEST, imm8)
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PSRLW __m128i _mm_srli_epi16 (__m128i m, int count)

PSRLW __m128i _mm_srl_epi16 (__m128i m, __m128i count)

PSRLD __m128i _mm_srli_epi32 (__m128i m, int count)

PSRLD __m128i _mm_srl_epi32 (__m128i m, __m128i count)

PSRLQ __m128i _mm_srli_epi64 (__m128i m, int count)

PSRLQ __m128i _mm_srl_epi64 (__m128i m, __m128i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4 and 7 for non-VEX-encoded instructions.

#UD	If VEX.L = 1.

PTEST- Packed Bit Test

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 17 /r PTEST xmm1, xmm2/m128	A	V/V	SSE4_1	Set ZF and CF depending on bit-wise AND and ANDN of sources
VEX.128.66.0F38.WIG 17 /r VPTEST xmm1, xmm2/m128	A	V/V	AVX	Set ZF and CF depending on bit-wise AND and ANDN of sources
VEX.256.66.0F38.WIG 17 /r VPTEST ymm1, ymm2/m256	A	V/V	AVX	Set ZF and CF depending on bit-wise AND and ANDN of sources
VEX.128.66.0F38.W0 0E /r VTESTPS xmm1, xmm2/m128	A	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources
VEX.256.66.0F38.W0 0E /r VTESTPS ymm1, ymm2/m256	A	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources
VEX.128.66.0F38.W0 0F /r VTESTPD xmm1, xmm2/m128	A	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources
VEX.256.66.0F38.W0 0F /r VTESTPD ymm1, ymm2/m256	A	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r)	ModRM:reg (r)	NA	NA

Description

PTEST and VPTEST set the ZF flag if all bits in the result are 0 of the bitwise AND of the first source operand (first operand) and the second source operand (second

operand). VPTEST sets the CF flag if all bits in the result are 0 of the bitwise AND of the second source operand (second operand) and the logical NOT of the destination operand.

VTESTPS performs a bitwise comparison of all the sign bits of the packed single-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND of the inverted source sign bits with the dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

VTESTPD performs a bitwise comparison of all the sign bits of the double-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND the inverted source sign bits with the dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

The first source register is specified by the ModR/M *reg* field.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

128-bit version: The first source register is an XMM register. The second source register can be an XMM register or a 256-bit memory location. The destination register is not modified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VPTEST (VEX.256 encoded version)

```
IF (SRC[255:0] BITWISE AND DEST[255:0] == 0) THEN ZF ← 1;
    ELSE ZF ← 0;
IF (SRC[255:0] BITWISE AND NOT DEST[255:0] == 0) THEN CF ← 1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

PTEST (128-bit versions)

```
IF (SRC[127:0] BITWISE AND DEST[127:0] == 0)
    THEN ZF ← 1;
    ELSE ZF ← 0;
IF (SRC[127:0] BITWISE AND NOT DEST[127:0] == 0)
    THEN CF ← 1;
    ELSE CF ← 0;
```

```
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

VTESTPS (VEX.256 encoded version)

```
TEMP[255:0] ← SRC[255:0] AND DEST[255:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127]= TEMP[160] =TEMP[191] = TEMP[224] =
TEMP[255] = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;

TEMP[255:0] ← SRC[255:0] AND NOT DEST[255:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127]= TEMP[160] =TEMP[191] = TEMP[224] =
TEMP[255] = 0)
    THEN CF ←1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

VTESTPD (VEX.256 encoded version)

```
TEMP[255:0] ← SRC[255:0] AND DEST[255:0]
IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;

TEMP[255:0] ← SRC[255:0] AND NOT DEST[255:0]
IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)
    THEN CF ←1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPTEST

int _mm256_testz_si256 (__m256i s1, __m256i s2);

int _mm256_testc_si256 (__m256i s1, __m256i s2);

int _mm256_testnzc_si256 (__m256i s1, __m256i s2);

int _mm_testz_si128 (__m128i s1, __m128i s2);

int _mm_testc_si128 (__m128i s1, __m128i s2);

int _mm_testnzc_si128 (__m128i s1, __m128i s2);

VTESTPS

int _mm256_testz_ps (__m256 s1, __m256 s2);

int _mm256_testc_ps (__m256 s1, __m256 s2);

int _mm256_testnzc_ps (__m256 s1, __m128 s2);

int _mm_testz_ps (__m128 s1, __m128 s2);

int _mm_testc_ps (__m128 s1, __m128 s2);

int _mm_testnzc_ps (__m128 s1, __m128 s2);

VTESTPD

int _mm256_testz_pd (__m256d s1, __m256d s2);

int _mm256_testc_pd (__m256d s1, __m256d s2);

int _mm256_testnzc_pd (__m256d s1, __m256d s2);

int _mm_testz_pd (__m128d s1, __m128d s2);

int _mm_testc_pd (__m128d s1, __m128d s2);

int _mm_testnzc_pd (__m128d s1, __m128d s2);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv != 1111.
	If VEX.W =1 for VTESTPS or VTESTPD.

PSUBB/PSUBW/PSUBD/PSUBQ -Packed Integer Subtract

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F F8 /r PSUBB xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed byte integers in xmm2/m128 from xmm1.
66 0F F9 /r PSUBW xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed word integers in xmm2/m128 from xmm1.
66 0F FA /r PSUBD xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed doubleword integers in xmm2/m128 from xmm1.
66 0F FB/r PSUBQ xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed quadword integers in xmm2/m128 from xmm1.
VEX.NDS.128.66.0F.WIG F8 /r VPSUBB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed byte integers in xmm3/m128 from xmm2.
VEX.NDS.128.66.0F.WIG F9 /r VPSUBW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed word integers in xmm3/m128 from xmm2.
VEX.NDS.128.66.0F.WIG FA /r VPSUBD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed doubleword integers in xmm3/m128 from xmm2.
VEX.NDS.128.66.0F.WIG FB/r VPSUBQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed quadword integers in xmm3/m128 from xmm2.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Subtracts the packed byte, word, doubleword, or quadword integers in the second source operand from the first source operand and stores the result in the destination operand. The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. When a result is too large to be represented in the 8/16/32/64 integer (overflow), the result is wrapped around and the low bits are written to the destination element (that is, the carry is ignored).

Note that these instructions can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

VPSUBB (VEX.128 encoded version)

DEST[7:0] ← SRC1[7:0]-SRC2[7:0]
DEST[15:8] ← SRC1[15:8]-SRC2[15:8]
DEST[23:16] ← SRC1[23:16]-SRC2[23:16]
DEST[31:24] ← SRC1[31:24]-SRC2[31:24]
DEST[39:32] ← SRC1[39:32]-SRC2[39:32]
DEST[47:40] ← SRC1[47:40]-SRC2[47:40]
DEST[55:48] ← SRC1[55:48]-SRC2[55:48]
DEST[63:56] ← SRC1[63:56]-SRC2[63:56]
DEST[71:64] ← SRC1[71:64]-SRC2[71:64]
DEST[79:72] ← SRC1[79:72]-SRC2[79:72]
DEST[87:80] ← SRC1[87:80]-SRC2[87:80]
DEST[95:88] ← SRC1[95:88]-SRC2[95:88]
DEST[103:96] ← SRC1[103:96]-SRC2[103:96]
DEST[111:104] ← SRC1[111:104]-SRC2[111:104]
DEST[119:112] ← SRC1[119:112]-SRC2[119:112]
DEST[127:120] ← SRC1[127:120]-SRC2[127:120]
DEST[255:128] ← 0

PSUBB (128-bit Legacy SSE version)
DEST[7:0] ← DEST[7:0]-SRC[7:0]
DEST[15:8] ← DEST[15:8]-SRC[15:8]
DEST[23:16] ← DEST[23:16]-SRC[23:16]
DEST[31:24] ← DEST[31:24]-SRC[31:24]
DEST[39:32] ← DEST[39:32]-SRC[39:32]
DEST[47:40] ← DEST[47:40]-SRC[47:40]
DEST[55:48] ← DEST[55:48]-SRC[55:48]
DEST[63:56] ← DEST[63:56]-SRC[63:56]
DEST[71:64] ← DEST[71:64]-SRC[71:64]
DEST[79:72] ← DEST[79:72]-SRC[79:72]
DEST[87:80] ← DEST[87:80]-SRC[87:80]
DEST[95:88] ← DEST[95:88]-SRC[95:88]
DEST[103:96] ← DEST[103:96]-SRC[103:96]
DEST[111:104] ← DEST[111:104]-SRC[111:104]
DEST[119:112] ← DEST[119:112]-SRC[119:112]
DEST[127:120] ← DEST[127:120]-SRC[127:120]
DEST[255:128] (Unmodified)

VPSUBW (VEX.128 encoded version)
DEST[15:0] ← SRC1[15:0]-SRC2[15:0]
DEST[31:16] ← SRC1[31:16]-SRC2[31:16]
DEST[47:32] ← SRC1[47:32]-SRC2[47:32]
DEST[63:48] ← SRC1[63:48]-SRC2[63:48]
DEST[79:64] ← SRC1[79:64]-SRC2[79:64]
DEST[95:80] ← SRC1[95:80]-SRC2[95:80]
DEST[111:96] ← SRC1[111:96]-SRC2[111:96]
DEST[127:112] ← SRC1[127:112]-SRC2[127:112]
DEST[255:128] ← 0

PSUBW (128-bit Legacy SSE version)
DEST[15:0] ← DEST[15:0]-SRC[15:0]
DEST[31:16] ← DEST[31:16]-SRC[31:16]
DEST[47:32] ← DEST[47:32]-SRC[47:32]
DEST[63:48] ← DEST[63:48]-SRC[63:48]
DEST[79:64] ← DEST[79:64]-SRC[79:64]
DEST[95:80] ← DEST[95:80]-SRC[95:80]
DEST[111:96] ← DEST[111:96]-SRC[111:96]
DEST[127:112] ← DEST[127:112]-SRC[127:112]
DEST[255:128] (Unmodified)

VPSUBD (VEX.128 encoded version)
DEST[31:0] ← SRC1[31:0]-SRC2[31:0]

DEST[63:32] ← SRC1[63:32]-SRC2[63:32]
DEST[95:64] ← SRC1[95:64]-SRC2[95:64]
DEST[127:96] ← SRC1[127:96]-SRC2[127:96]
DEST[255:128] ← 0

PSUBD (128-bit Legacy SSE version)
DEST[31:0] ← DEST[31:0]-SRC[31:0]
DEST[63:32] ← DEST[63:32]-SRC[63:32]
DEST[95:64] ← DEST[95:64]-SRC[95:64]
DEST[127:96] ← DEST[127:96]-SRC[127:96]
DEST[255:128] (Unmodified)

VPSUBQ (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0]-SRC2[63:0]
DEST[127:64] ← SRC1[127:64]-SRC2[127:64]
DEST[255:128] ← 0

PSUBQ (128-bit Legacy SSE version)
DEST[63:0] ← DEST[63:0]-SRC[63:0]
DEST[127:64] ← DEST[127:64]-SRC[127:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PSUBB __m128i _mm_sub_epi8 (__m128i a, __m128i b)

PSUBW __m128i _mm_sub_epi16 (__m128i a, __m128i b)

PSUBD __m128i _mm_sub_epi32 (__m128i a, __m128i b)

PSUBQ __m128i _mm_sub_epi64(__m128i m1, __m128i m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1

PSUBSB/PSUBSW -Subtract Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E8 /r PSUBSB xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed signed byte integers in xmm2/m128 from packed signed byte integers in xmm1 and saturate results.
66 0F E9 /r PSUBSW xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed signed word integers in xmm2/m128 from packed signed word integers in xmm1 and saturate results.
VEX.NDS.128.66.0F.WIG E8 /r VPSUBSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results.
VEX.NDS.128.66.0F.WIG E9 /r VPSUBSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the packed signed integers of the second source operand from the packed signed integers of the first source operand, and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

The first source and destination operands are XMM registers and the second source operand is either an XMM register or a 128-bit memory location.

The PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The PSUBSW instruction subtracts packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

VPSUBSB

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[255:128] ← 0
```

PSUBSB

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[127:120] - SRC[127:120]);
DEST[255:128] (Unmodified)
```

VPSUBSW

```
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[255:128] ← 0
```

PSUBSW

```
DEST[15:0] ← SaturateToSignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (DEST[127:112] - SRC[127:112]);
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

PSUBSB __m128i _mm_subs_epi8(__m128i m1, __m128i m2)

PSUBSW __m128i _mm_subs_epi16(__m128i m1, __m128i m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	if VEX.L = 1.

PSUBUSB/PSUBUSW -Subtract Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F D8 /r PSUBUSB xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed unsigned byte integers in xmm2/m128 from packed unsigned byte integers in xmm1 and saturate result.
66 0F D9 /r PSUBUSW xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed unsigned word integers in xmm2/m128 from packed unsigned word integers in xmm1 and saturate result.
VEX.NDS.128.66.0F.WIG D8 /r VPSUBUSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed unsigned byte integers in xmm3/m128 from packed unsigned byte integers in xmm2 and saturate result.
VEX.NDS.128.66.0F.WIG D9 /r VPSUBUSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed unsigned word integers in xmm3/m128 from packed unsigned word integers in xmm2 and saturate result.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the packed unsigned integers of the second source operand from the packed unsigned integers of the first source operand and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

The first source and destination operands are XMM registers. The second source operand can be either an XMM register or a 128-bit memory location.

The PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

VPSUBUSB

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[255:128] ← 0
```

PSUBUSB

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (DEST[127:120] - SRC[127:120]);
DEST[255:128] (Unmodified)
```

VPSUBUSW

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[255:128] ← 0
```

PSUBUSW

```
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (DEST[127:112] - SRC[127:112]);
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

PSUBUSB __m128i _mm_subs_epu8(__m128i m1, __m128i m2)

PSUBUSW __m128i _mm_subs_epu16(__m128i m1, __m128i m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ - Unpack High Data

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 68/r PUNPCKHBW xmm1,xmm2/m128	A	V/V	SSE2	Interleave high-order bytes from xmm1 and xmm2/m128 into xmm1.
66 0F 69/r PUNPCKHWD xmm1,xmm2/m128	A	V/V	SSE2	Interleave high-order words from xmm1 and xmm2/m128 into xmm1.
66 0F 6A/r PUNPCKHDQ xmm1, xmm2/m128	A	V/V	SSE2	Interleave high-order doublewords from xmm1 and xmm2/m128 into xmm1.
66 0F 6D/r PUNPCKHQDQ xmm1, xmm2/m128	A	V/V	SSE2	Interleave high-order quadword from xmm1 and xmm2/m128 into xmm1 register.
VEX.NDS.128.66.0F.WIG 68/r VPUNPCKHBW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 69/r VPUNPCKHWD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order words from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 6A/r VPUNPCKHDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order doublewords from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 6D/r VPUNPCKHQDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order quadword from xmm2 and xmm3/m128 into xmm1 register.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, and quadwords) of the first source operand and second source operand into the destination operand. (Figure 5-24 shows the unpack operation for bytes in 64-bit operands.). The low-order data elements are ignored.

128-bit Legacy SSE version: The first source operand and the destination operand are the same.

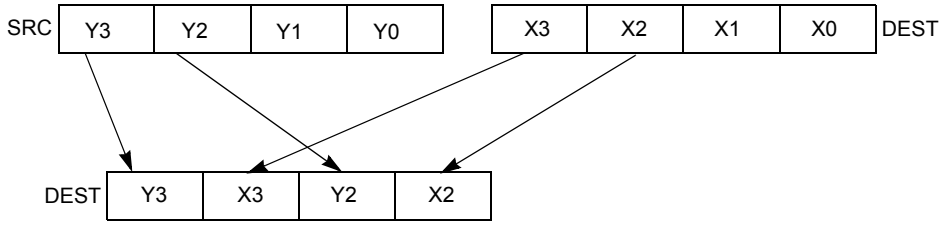


Figure 5-24. PUNPCKHDQ Instruction Operation

The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the PUNPCKHDQ instruction interleaves the high order doubleword (or doublewords) of the source and destination operands, and the PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

128-bit Legacy SSE versions: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

INTERLEAVE_HIGH_BYTES (SRC1, SRC2)
DEST[7:0] ← SRC1[71:64]
DEST[15:8] ← SRC2[71:64]
DEST[23:16] ← SRC2[79:72]
DEST[31:24] ← SRC2[79:72]
DEST[39:32] ← SRC1[87:80]
DEST[47:40] ← SRC2[87:80]
DEST[55:48] ← SRC1[95:88]
DEST[63:56] ←SRC2[95:88]
DEST[71:64] ← SRC1[103:96]
DEST[79:72] ← SRC2[103:96]
DEST[87:80] ← SRC1[111:104]
DEST[95:88] ← SRC2[111:104]
DEST[103:96] ← SRC1[119:112]
DEST[111:104] ← SRC2[119:112]
DEST[119:112] ← SRC1[127:120]
DEST[127:120] ← SRC2[127:120]

INTERLEAVE_HIGH_WORDS (SRC1, SRC2)
DEST[15:0] ← SRC1[79:64]
DEST[31:16] ← SRC2[79:64]
DEST[47:32] ← SRC1[95:80]
DEST[63:48] ← SRC2[95:80]
DEST[79:64] ← SRC1[111:96]
DEST[95:80] ← SRC2[111:96]
DEST[111:96] ← SRC1[127:112]
DEST[127:112] ← SRC2[127:112]

INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]

INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]

PUNPCKHBW
DEST[127:0] ← INTERLEAVE_HIGH_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKHBW
DEST[127:0] ← INTERLEAVE_HIGH_BYTES(SRC1, SRC2)
DEST[255:127] ← 0

PUNPCKHWD
DEST[127:0] ← INTERLEAVE_HIGH_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKHWD
DEST[127:0] ← INTERLEAVE_HIGH_WORDS(SRC1, SRC2)
DEST[255:127] ← 0

PUNPCKHDQ
DEST[127:0] ← INTERLEAVE_HIGH_DWORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKHDQ
DEST[127:0] ← INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[255:127] ← 0

PUNPCKHQDQ
DEST[127:0] ← INTERLEAVE_HIGH_QWORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKHQDQ
DEST[127:0] ← INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)

DEST[255:127] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PUNPCKHBW __m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)

PUNPCKHWD __m128i _mm_unpackhi_epi16(__m128i m1,__m128i m2)

PUNPCKHDQ __m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)

PUNPCKHQDQ __m128i _mm_unpackhi_epi64 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ - Unpack Low Data

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 60/r PUNPCKLBW xmm1,xmm2/m128	A	V/V	SSE2	Interleave low-order bytes from xmm1 and xmm2/m128 into xmm1.
66 0F 61/r PUNPCKLWD xmm1,xmm2/m128	A	V/V	SSE2	Interleave low-order words from xmm1 and xmm2/m128 into xmm1.
66 0F 62/r PUNPCKLDQ xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order doublewords from xmm1 and xmm2/m128 into xmm1.
66 0F 6C/r PUNPCKLQDQ xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order quadword from xmm1 and xmm2/m128 into xmm1 register.
VEX.NDS.128.66.0F.WIG 60/r VPUNPCKLBW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 61/r VPUNPCKLWD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order words from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 62/r VPUNPCKLDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order doublewords from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 6C/r VPUNPCKLQDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order quadword from xmm2 and xmm3/m128 into xmm1 register.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the first source operand and second source operand into the destination operand. (Figure 5-25 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

128-bit Legacy SSE version: The first source operand and the destination operand are the same.

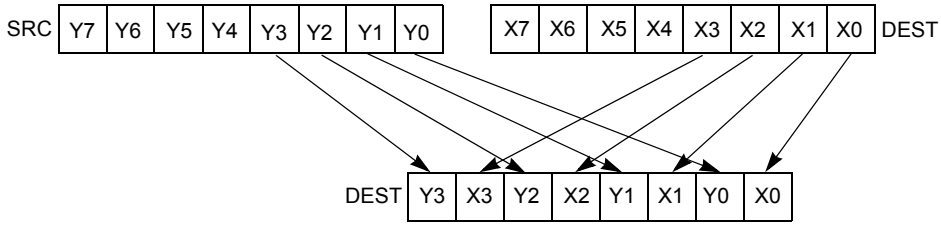


Figure 5-25. PUNPCKLBW Instruction Operation using 64-bit Operands

The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the PUNPCKLDQ instruction interleaves the low order doubleword (or doublewords) of the source and destination operands, and the PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

128-bit Legacy SSE versions: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

INTERLEAVE_BYTES (SRC1, SRC2)

DEST[7:0] ← SRC1[7:0]

DEST[15:8] ← SRC2[7:0]
DEST[23:16] ← SRC2[15:8]
DEST[31:24] ← SRC2[15:8]
DEST[39:32] ← SRC1[23:16]
DEST[47:40] ← SRC2[23:16]
DEST[55:48] ← SRC1[31:24]
DEST[63:56] ←SRC2[31:24]
DEST[71:64] ← SRC1[39:32]
DEST[79:72] ← SRC2[39:32]
DEST[87:80] ← SRC1[47:40]
DEST[95:88] ← SRC2[47:40]
DEST[103:96] ← SRC1[55:48]
DEST[111:104] ← SRC2[55:48]
DEST[119:112] ← SRC1[63:56]
DEST[127:120] ← SRC2[63:56]

INTERLEAVE_WORDS (SRC1, SRC2)
DEST[15:0] ← SRC1[15:0]
DEST[31:16] ← SRC2[15:0]
DEST[47:32] ← SRC1[31:16]
DEST[63:48] ← SRC2[31:16]
DEST[79:64] ← SRC1[47:32]
DEST[95:80] ← SRC2[47:32]
DEST[111:96] ← SRC1[63:48]
DEST[127:112] ← SRC2[63:48]

INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]

INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]

PUNPCKLBW
DEST[127:0] ← INTERLEAVE_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLBW
DEST[127:0] ← INTERLEAVE_BYTES(SRC1, SRC2)
DEST[255:127] ← 0

PUNPCKLWD
DEST[127:0] ← INTERLEAVE_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLWD
DEST[127:0] ← INTERLEAVE_WORDS(SRC1, SRC2)
DEST[255:127] ← 0

PUNPCKLDQ
DEST[127:0] ← INTERLEAVE_DWORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLDQ
DEST[127:0] ← INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[255:127] ← 0

PUNPCKLQDQ
DEST[127:0] ← INTERLEAVE_QWORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLQDQ
DEST[127:0] ← INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[255:127] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PUNPCKLBW __m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)

PUNPCKLWD __m128i _mm_unpacklo_epi16 (__m128i m1, __m128i m2)

PUNPCKLDQ __m128i _mm_unpacklo_epi32 (__m128i m1, __m128i m2)

PUNPCKLQDQ __m128i _mm_unpacklo_epi64 (__m128i m1, __m128i m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PXOR - Exclusive Or

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F EF /r PXOR xmm1, xmm2/m128	A	V/V	SSE2	Bitwise XOR of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG EF /r VPXOR xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise XOR of xmm3/m128 and xmm2.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical XOR operation on the second source operand and the first source operand and stores the result in the destination operand. The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are different; otherwise, each bit is 0 if the corresponding bits of the first and second operand are the same.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

VPXOR (VEX.128 encoded version)

DEST ← SRC1 XOR SRC2
DEST[255:128] ← 0

PXOR (128-bit Legacy SSE version)

DEST ← DEST XOR SRC
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PXOR __m128i _mm_xor_si128 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

RCPPS- Compute Approximate Reciprocals of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 53 /r RCPPS xmm1, xmm2/m128	A	V/V	SSE	Computes the approximate reciprocals of packed single-precision values in xmm2/mem and stores the results in xmm1
VEX.128.0F.WIG 53 /r VRCPPS xmm1, xmm2/m128	A	V/V	AVX	Computes the approximate reciprocals of packed single-precision values in xmm2/mem and stores the results in xmm1
VEX.256.0F.WIG 53 /r VRCPPS ymm1, ymm2/m256	A	V/V	AVX	Computes the approximate reciprocals of packed single-precision values in ymm2/mem and stores the results in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Performs an SIMD computation of the approximate reciprocals of the four or eight packed single precision floating-point values in the source operand (second operand) and stores the packed single-precision floating-point results in the destination operand. See Figure 10-5 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1 for an illustration of an SIMD single-precision floating-point operation.

The relative error for this approximation is:

$|\text{Relative Error}| < 1.5 * 2^{-12}$

The RCPPS instruction is not affected by the rounding control bits in the MXCSR register.

When a source value is a 0.0, an Inf of the sign of the source value is returned.

A denormal source value is treated as a 0.0 (of the same sign).

Tiny results are always flushed to 0.0, with the sign of the operand:

- The result is guaranteed not to be tiny for inputs that are not greater than $(2^{125})*(2-3*2^{-10})$ in absolute value.
- The result is guaranteed to be flushed to 0 for values greater than $(2^{126})*(1+3*2^{-11})$ in absolute value.
- Input values in between this range may or may not produce tiny results, depending on the implementation.

When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VRCPPS (VEX.256 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[159:128] ← APPROXIMATE(1/SRC[159:128])
DEST[191:160] ← APPROXIMATE(1/SRC[191:160])
DEST[223:192] ← APPROXIMATE(1/SRC[223:192])
DEST[255:224] ← APPROXIMATE(1/SRC[255:224])
```

VRCPPS (VEX.128 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[255:128] ← 0
```

RCPPS (128-bit Legacy SSE version)

DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

RCPPS __m256 _mm256_rcp_ps (__m256 a);

RCPPS __m128 _mm_rcp_ps (__m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv != 1111B.

RCPSS - Compute Reciprocal of Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 53 /r RCPSS xmm1, xmm2/m32	A	V/V	SSE	Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm2/m32 and stores the result in xmm1.
VEX.NDS.LIG.F3.0F.WIG 53 /r VRCPSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm3/m32 and stores the result in xmm1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes of an approximate reciprocal of the low single-precision floating-point value in the second source operand and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source operand and the destination operand are XMM registers. The three high-order doublewords of the destination operand are copied from the same bits of the first source operand. See Figure 10-6 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$|\text{Relative Error}| < 1.5 * 2^{-12}$

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an Inf of the sign of the source value is returned.

A denormal source value is treated as a 0.0 (of the same sign).

Tiny results are always flushed to 0.0, with the sign of the operand:

- The result is guaranteed not to be tiny for inputs that are not greater than $(2^{125})*(2-3*2^{-10})$ in absolute value.
- The result is guaranteed to be flushed to 0 for values greater than $(2^{126})*(1+3*2^{-11})$ in absolute value.
- Input values in between this range may or may not produce tiny results, depending on the implementation.

When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VRCPSS (VEX.128 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SRC2[31:0])
DEST[127:32] ← SRC1[127:32]
DEST[255:128] ← 0
```

RCPSS (128-bit Legacy SSE version)

```
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[255:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

RCPSS __m128 _mm_rcp_ss(__m128 a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5

RSQRTPS - Compute Approximate Reciprocals of Square Roots of Packed Single-Precision Floating-point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 52 /r RSQRTPS xmm1, xmm2/m128	A	V/V	SSE	Computes the approximate reciprocals of the square roots of packed single-precision values in xmm2/mem and stores the results in xmm1
VEX.128.0F.WIG 52 /r VRSQRTPS xmm1, xmm2/m128	A	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single-precision values in xmm2/mem and stores the results in xmm1
VEX.256.0F.WIG 52 /r VRSQRTPS ymm1, ymm2/m256	A	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single-precision values in ymm2/mem and stores the results in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Performs an SIMD computation of the approximate reciprocals of the square roots of the four or eight packed single precision floating-point values in the source operand (second operand) and stores the packed single-precision floating-point results in the destination operand. See Figure 10-5 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1 for an illustration of an SIMD single-precision floating-point operation.

$|\text{Relative Error}| < 1.5 * 2^{-12}$

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register.

When a source value is a 0.0, an Inf of the sign of the source value is returned.

A denormal source value is treated as a 0.0 (of the same sign).

When a source value is a negative value (other than 0.0), a floating-point indefinite is returned.

When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VRSQRTPS (VEX.256 encoded version)

DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[159:128] ← APPROXIMATE(1/SQRT(SRC2[159:128]))
DEST[191:160] ← APPROXIMATE(1/SQRT(SRC2[191:160]))
DEST[223:192] ← APPROXIMATE(1/SQRT(SRC2[223:192]))
DEST[255:224] ← APPROXIMATE(1/SQRT(SRC2[255:224]))

VRSQRTPS (VEX.128 encoded version)

DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[255:128] ← 0

RSQRTPS (128-bit Legacy SSE version)

DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

RSQRTPS __m256 _mm256_rsqrt_ps (__m256 a);
RSQRTPS __m128 _mm_rsqrt_ps (__m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv != 1111B.

RSQRTSS - Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 52 /r RSQRTSS xmm1, xmm2/m32	A	V/V	SSE	Computes the approximate reciprocal of the square root of the low single precision floating-point value in xmm2/m32 and stores the results in xmm1.
VEX.NDS.LIG.F3.0F.WIG 52 /r VRSQRTSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Computes the approximate reciprocal of the square root of the low single precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes an approximate reciprocal of the square root of the low single-precision floating-point value in the second source operand stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. The three high-order doublewords of the destination operand are copied from the same bits of the first source operand. See *Figure 10-6* in the *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating point operation. The relative error for this approximation is:

$|\text{Relative Error}| < 1.5 * 2^{-12}$

The RSQRTSS instruction is not affected by the rounding control bits in the MXCSR register.

When a source value is a 0.0, an Inf of the sign of the source value is returned.

A denormal source value is treated as a 0.0 (of the same sign).

When a source value is a negative value (other than 0.0), a floating-point indefinite is returned.

When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VRSQRTSS (VEX.128 encoded version)

DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))
DEST[127:32] ← SRC1[31:0]
DEST[255:128] ←0

RSQRTSS (128-bit Legacy SSE version)

DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))
DEST[255:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

RSQRTSS __m128 _mm_rsqrt_ss(__m128 a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5

ROUNDPD- Round Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 09 /r ib ROUNDPD xmm1, xmm2/m128, imm8	A	V/V	SSE4_1	Round packed double-precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8
VEX.128.66.0F3A.WIG 09 /r ib VROUNDPD xmm1, xmm2/m128, imm8	A	V/V	AVX	Round packed double-precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8
VEX.256.66.0F3A.WIG 09 /r ib VROUNDPD ymm1, ymm2/m256, imm8	A	V/V	AVX	Round packed double-precision floating-point values in ymm2/m256 and place the result in ymm1. The rounding mode is determined by imm8

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Round the four double-precision floating-point values in the source operand (second operand) by the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds the input to an integral value and returns the result as a double-precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 5-26. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Figure 5-26 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

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

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

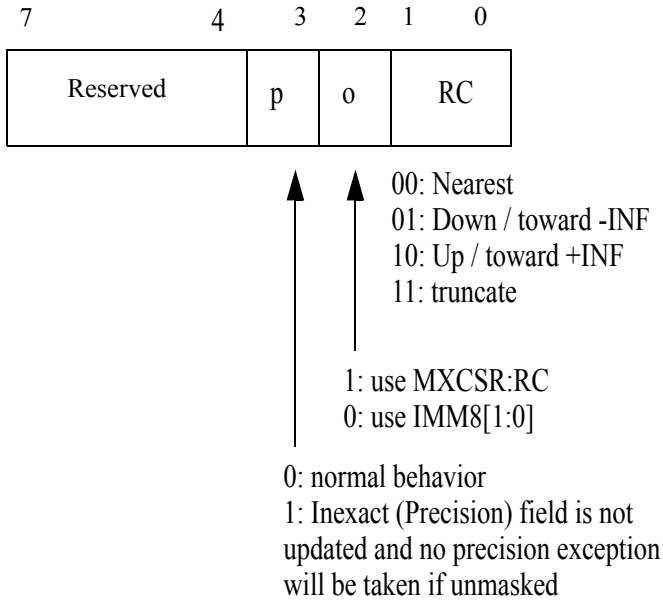


Figure 5-26. VROUNDxx immediate control field definition

Operation

```
RoundToInteger(value, control) {
    rounding_direction ← MXCSR:RC
    if (control[2] == 1)
        rounding_direction ← MXCSR:RC
    else
        rounding_direction ← control[1:0]
    fi
    case (rounding_direction)
        00: dest ← round_to_nearest_even_integer(value)
        01: dest ← round_to_equal_or_smaller_integer(value)
        10: dest ← round_to_equal_or_larger_integer(value)
        11: dest ← round_to_nearest_smallest_magnitude_integer(value)
    esac


    if (control[3] = 0)
    {
        if (value != dest)
        {
            set_precision()
        }
    }
    return(dest)
}
```

VROUNDPD (VEX.256 encoded version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64]], ROUND_CONTROL)
DEST[191:128] ← RoundToInteger(SRC[191:128]], ROUND_CONTROL)
DEST[255:192] ← RoundToInteger(SRC[255:192] ], ROUND_CONTROL)
```

VROUNDPD (VEX.128 encoded version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0]], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64]], ROUND_CONTROL)
DEST[255:128] ← 0
```

ROUNDPD (128-bit Legacy SSE version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0]], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64]], ROUND_CONTROL)
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

__m256 _mm256_round_pd(__m256d s1, int iRoundMode);

__m256 _mm256_floor_pd(__m256d s1);

__m256 _mm256_ceil_pd(__m256d s1)

__m128 _mm_round_pd(__m128d s1, int iRoundMode);

__m128 _mm_floor_pd(__m128d s1);

__m128 _mm_ceil_pd(__m128d s1)

SIMD Floating-Point Exceptions

Precision, Invalid

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

ROUNDPS- Round Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 08 /r ib ROUNDPS xmm1, xmm2/m128, imm8	A	V/V	SSE4_1	Round packed single-precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8
VEX.128.66.0F3A.WIG 08 /r ib VROUNDPS xmm1, xmm2/m128, imm8	A	V/V	AVX	Round packed single-precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8
VEX.256.66.0F3A.WIG 08 /r ib VROUNDPS ymm1, ymm2/m256, imm8	A	V/V	AVX	Round packed single-precision floating-point values in ymm2/m256 and place the result in ymm1. The rounding mode is determined by imm8

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Round the four or eight single-precision floating-point values in the source operand (second operand) by the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds the input to an integral value and returns the result as a single-precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 5-26. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Figure 5-26 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

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

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

(see ROUNDPD for definition of RoundToInteger)

VROUNDPS (VEX.256 encoded version)

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64]], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96]], ROUND_CONTROL)
DEST[159:128] ← RoundToInteger(SRC[159:128]], ROUND_CONTROL)
DEST[191:160] ← RoundToInteger(SRC[191:160]], ROUND_CONTROL)
DEST[223:192] ← RoundToInteger(SRC[223:192]], ROUND_CONTROL)
DEST[255:224] ← RoundToInteger(SRC[255:224]], ROUND_CONTROL)

VROUNDPS (VEX.128 encoded version)

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64]], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96]], ROUND_CONTROL)
DEST[255:128] ← 0

ROUNDPS(128-bit Legacy SSE version)

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64]], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96]], ROUND_CONTROL)
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

__m256 _mm256_round_ps(__m256 s1, int iRoundMode);

__m256 _mm256_floor_ps(__m256 s1);

__m256 _mm256_ceil_ps(__m256 s1)

__m128 _mm_round_ps(__m128 s1, int iRoundMode);

__m128 _mm_floor_ps(__m128 s1);

__m128 _mm_ceil_ps(__m128 s1)

SIMD Floating-Point Exceptions

Precision, Invalid

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

ROUNDSD - Round Scalar Double-Precision Value

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0B /r ib ROUNDSD xmm1, xmm2/m64, imm8	A	V/V	SSE4_1	Round the low packed double precision floating-point value in xmm2/m64 and place the result in xmm1. The rounding mode is determined by imm8.
VEX.NDS.LIG.66.0F3A.WIG 0B /r ib VROUNDSD xmm1, xmm2, xmm3/m64, imm8	B	V/V	AVX	Round the low packed double precision floating-point value in xmm3/m64 and place the result in xmm1. The rounding mode is determined by imm8. Upper packed double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Round the DP FP value in the second source operand by the rounding mode specified in the immediate operand and place the result in the destination operand. The rounding process rounds the lowest double precision floating-point input to an integral value and returns the result as a double precision floating-point value in the lowest position. The upper double precision floating-point value in the destination is retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 5-26. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Figure 5-26 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VROUNDSD (VEX.128 encoded version)

DEST[63:0] ← RoundToInteger(SRC2[63:0], ROUND_CONTROL)
DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0

ROUNDSD (128-bit Legacy SSE version)

DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[255:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

ROUNDSD __m128d _mm_round_sd(__m128d dst, __m128d s1, int iRoundMode);

__m128d _mm_floor_sd(__m128d dst, __m128d s1);

__m128d _mm_ceil_sd(__m128d dst, __m128d s1);

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN), Precision (signaled only if imm[3] == '0; if imm[3] == '1, then the Precision Mask in the MXSCSR is ignored.)

Note that Denormal is not signaled by ROUNDSD.

Other Exceptions

See Exceptions Type 3

ROUNDSS - Round Scalar Single-Precision Value

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0A ib ROUNDSS xmm1, xmm2/m32, imm8	A	V/V	SSE4_1	Round the low packed single precision floating-point value in xmm2/m32 and place the result in xmm1. The rounding mode is determined by imm8.
VEX.NDS.LIG.66.0F3A.WIG 0A ib VROUNDSS xmm1, xmm2, xmm3/m32, imm8	B	V/V	AVX	Round the low packed single precision floating-point value in xmm3/m32 and place the result in xmm1. The rounding mode is determined by imm8. Also, upper packed single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Round the single precision floating-point value in the second source operand by the rounding mode specified in the immediate operand and place the result in the destination operand. The rounding process rounds the lowest single precision floating-point input to an integral value and returns the result as a single precision floating-point value in the lowest position. The upper three single precision floating-point values in the destination are copied from the first source operand.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 5-26. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Figure 5-26 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VROUNDSS (VEX.128 encoded version)

```
DEST[31:0] ← RoundToInteger(SRC2[31:0], ROUND_CONTROL)
DEST[127:32] ← SRC1[127:32]
DEST[255:128] ← 0
```

ROUNDSS (128-bit Legacy SSE version)

```
DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[255:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

ROUNDSS __m128 _mm_round_ss(__m128 dst, __m128 s1, int iRoundMode);

__m128 _mm_floor_ss(__m128 dst, __m128 s1);

__m128 _mm_ceil_ss(__m128 dst, __m128 s1);

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN), Precision (signaled only if imm[3] == ‘0; if imm[3] == ‘1, then the Precision Mask in the MXSCSR is ignored.)

Note that Denormal is not signaled by ROUNDSS.

Other Exceptions

See Exceptions Type 3

SHUFPD - Shuffle Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C6 /r ib SHUFPD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle Packed double-precision floating-point values selected by imm8 from xmm1 and xmm2/mem
VEX.NDS.128.66.0F.WIG C6 /r ib VSHUFPD xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Shuffle Packed double-precision floating-point values selected by imm8 from xmm2 and xmm3/mem
VEX.NDS.256.66.0F.WIG C6 /r ib VSHUFPD ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Shuffle Packed double-precision floating-point values selected by imm8 from ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Moves either of the two packed double-precision floating-point values from each double quadword in the first source operand (second operand) into the low quadword of each double quadword of the destination operand (first operand); moves either of the two packed double-precision floating-point values from the second source operand (third operand) into the high quadword of each double quadword of the destination operand (see Figure 5-27). The immediate determines which values are moved to the destination operand.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

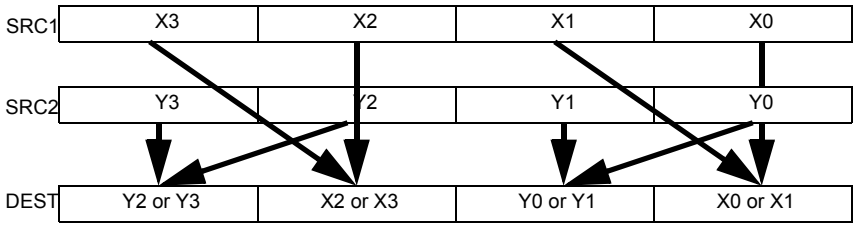


Figure 5-27. VSHUFPD Operation

Operation

VSHUFPD (VEX.256 encoded version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] FI;
IF IMM0[2] = 0
    THEN DEST[191:128] ← SRC1[191:128]
    ELSE DEST[191:128] ← SRC1[255:192] FI;
IF IMM0[3] = 0
    THEN DEST[255:192] ← SRC2[191:128]
    ELSE DEST[255:192] ← SRC2[255:192] FI;
```

VSHUFPD (VEX.128 encoded version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] FI;
DEST[255:128] ← 0
```

VSHUFPD (128-bit Legacy SSE version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] FI;
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFPD __m256d _mm256_shuffle_pd (__m256d a, __m256d b, const int select);
SHUFPD __m128d _mm_shuffle_pd (__m128d a, __m128d b, const int select);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

SHUFPS - Shuffle Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C6 /r ib SHUFPS xmm1, xmm3/m128, imm8	A	V/V	SSE	Shuffle Packed single-precision floating-point values selected by imm8 from xmm1 and xmm2/mem
VEX.NDS.128.0F.WIG C6 /r ib VSHUFPS xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Shuffle Packed single-precision floating-point values selected by imm8 from xmm2 and xmm3/mem
VEX.NDS.256.0F.WIG C6 /r ib VSHUFPS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Shuffle Packed single-precision floating-point values selected by imm8 from ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Moves two of the four packed single-precision floating-point values from each double qword of the first source operand (second operand) into the low quadword of each double qword of the destination operand (first operand); moves two of the four packed single-precision floating-point values from each double qword of the second source operand (third operand) into to the high quadword of each double qword of the destination operand (see Figure 5-28). The selector operand (third operand) determines which values are moved to the destination operand.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

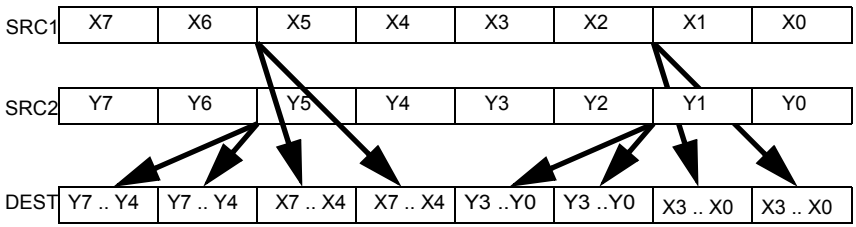


Figure 5-28. VSHUFPS Operation

Operation

```
Select4(SRC, control) {
CASE (control[1:0]) OF
    0:    TMP ← SRC[31:0];
    1:    TMP ← SRC[63:32];
    2:    TMP ← SRC[95:64];
    3:    TMP ← SRC[127:96];
ESAC;
RETURN TMP
}
```

VSHUFPS (VEX.256 encoded version)

```
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] ← Select4(SRC2[255:128], imm8[5:4]);
DEST[255:224] ← Select4(SRC2[255:128], imm8[7:6]);
```

VSHUFPS (VEX.128 encoded version)

```
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
```

DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
DEST[255:128] ← 0

SHUFPS (128-bit Legacy SSE version)
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFPS __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int select);
SHUFPS __m128 _mm_shuffle_ps (__m128 a, __m128 b, const int select);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

SQRTPD- Square Root of Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 51 /r SQRTPD xmm1, xmm2/m128	A	V/V	SSE2	Computes Square Roots of the packed double-precision floating-point values in xmm2/m128 and stores the result in xmm1
VEX.128.66.0F.WIG 51 /r VSQRTPD xmm1, xmm2/m128	A	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in xmm2/m128 and stores the result in xmm1
VEX.256.66.0F.WIG 51/r VSQRTPD ymm1, ymm2/m256	A	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in ymm2/m256 and stores the result in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Performs an SIMD computation of the square roots of the two or four packed double-precision floating-point values in the source operand (second operand) stores the packed double-precision floating-point results in the destination operand.

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

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VSQRTPD (VEX.256 encoded version)

DEST[63:0] ← SQRT(SRC[63:0])
DEST[127:64] ← SQRT(SRC[127:64])
DEST[191:128] ← SQRT(SRC[191:128])
DEST[255:192] ← SQRT(SRC[255:192])

.

VSQRTPD (VEX.128 encoded version)

DEST[63:0] ← SQRT(SRC[63:0])
DEST[127:64] ← SQRT(SRC[127:64])
DEST[255:128] ← 0

SQRTPD (128-bit Legacy SSE version)

DEST[63:0] ← SQRT(SRC[63:0])
DEST[127:64] ← SQRT(SRC[127:64])
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

SQRTPD __m256d _mm256_sqrt_pd (__m256d a);

SQRTPD __m128d _mm_sqrt_pd (__m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

SQRTPS- Square Root of Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 51 /r SQRTPS xmm1, xmm2/m128	A	V/V	SSE	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1
VEX.128.0F.WIG 51 /r VSQRTPS xmm1, xmm2/m128	A	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1
VEX.256.0F.WIG 51/r VSQRTPS ymm1, ymm2/m256	A	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in ymm2/m256 and stores the result in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Performs an SIMD computation of the square roots of the four or eight packed single-precision floating-point values in the source operand (second operand) stores the packed double-precision floating-point results in the destination operand.

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

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VSQRTPS (VEX.256 encoded version)

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

VSQRTPS (VEX.128 encoded version)

DEST[31:0] ← SQRT(SRC[31:0])
DEST[63:32] ← SQRT(SRC[63:32])
DEST[95:64] ← SQRT(SRC[95:64])
DEST[127:96] ← SQRT(SRC[127:96])
DEST[255:128] ← 0

SQRTPS (128-bit Legacy SSE version)

DEST[31:0] ← SQRT(SRC[31:0])
DEST[63:32] ← SQRT(SRC[63:32])
DEST[95:64] ← SQRT(SRC[95:64])
DEST[127:96] ← SQRT(SRC[127:96])
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

SQRTPS __m256 _mm256_sqrt_ps (__m256 a);

SQRTPS __m128 _mm_sqrt_ps (__m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

SQRTSD - Compute Square Root of Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 51/ SQRTSD xmm1,xmm2/m64	A	V/V	SSE2	Computes square root of the low double-precision floating point value in xmm2/m64 and stores the results in xmm1.
VEX.NDS.LIG.F2.0F.WIG 51/ VSQRTSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Computes square root of the low double-precision floating point value in xmm3/m64 and stores the results in xmm2. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the square root of the low double-precision floating-point value in the second source operand and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a scalar double-precision floating-point operation.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VSQRTSD (VEX.128 encoded version)

DEST[63:0] ← SQRT(SRC2[63:0])

DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0

SQRTSD (128-bit Legacy SSE version)
DEST[63:0] ← SQRT(SRC[63:0])
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

SQRTSD __m128d _mm_sqrt_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

SQRTSS - Compute Square Root of Scalar Single-Precision Value

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 51 SQRTSS xmm1, xmm2/m32	A	V/V	SSE	Computes square root of the low single-precision floating-point value in xmm2/m32 and stores the results in xmm1.
VEX.NDS.LIG.F3.0F.WIG 51 VSQRTSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Computes square root of the low single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the square root of the low single-precision floating-point value in the second source operand and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands is an XMM register. The three high order doublewords of the destination operand remain unchanged. See Figure 10-6 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a scalar single-precision floating-point operation.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VSQRTSS (VEX.128 encoded version)

DEST[31:0] ← SQRT(SRC2[31:0])
DEST[127:32] ← SRC1[127:32]

DEST[255:128] ← 0

SQRTSS (128-bit Legacy SSE version)
DEST[31:0] ← SQRT(SRC2[31:0])
DEST[255:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

SQRTSS __m128 _mm_sqrt_ss(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VSTMXCSR—Store MXCSR Register State

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.LZ.0F.WIG AE /3 VSTMXCSR *m32*	A	V/V	AVX	Store contents of MXCSR register to *m32*.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (w)	NA	NA	NA

Description

Stores the contents of the MXCSR control and status register to the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the MXCSR register are stored as 0s.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

VEX.L must be 0, otherwise instructions will #UD.

Operation

```
m32 ← MXCSR;
```

Intel C/C++ Compiler Intrinsic Equivalent

_mm_getcsr(void)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 9

SUBPD- Subtract Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5C /r SUBPD xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed double-precision floating-point values in xmm2/mem from xmm1 and stores result in xmm1
VEX.NDS.128.66.0F.WIG 5C /r VSUBPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Subtract packed double-precision floating-point values in xmm3/mem from xmm2 and stores result in xmm1
VEX.NDS.256.66.0F.WIG 5C /r VSUBPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Subtract packed double-precision floating-point values in ymm3/mem from ymm2 and stores result in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD subtract of the four or eight packed double-precision floating-point values of the second Source operand from the first Source operand, and stores the packed double-precision floating-point results in the destination operand.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: T second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VSUBPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
DEST[127:64] ← SRC1[127:64] - SRC2[127:64]
DEST[191:128] ← SRC1[191:128] - SRC2[191:128]
DEST[255:192] ← SRC1[255:192] - SRC2[255:192]
.

VSUBPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
DEST[127:64] ← SRC1[127:64] - SRC2[127:64]
DEST[255:128] ← 0

SUBPD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] - SRC[63:0]
DEST[127:64] ← DEST[127:64] - SRC[127:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VSUBPD __m256d _mm256_sub_pd (__m256d a, __m256d b);

SUBPD __m128d _mm_sub_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

SUBPS- Subtract Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 5C /r SUBPS xmm1, xmm2/m128	A	V/V	SSE	Subtract packed single-precision floating-point values in xmm2/mem from xmm1 and stores result in xmm1
VEX.NDS.128.0F.WIG 5C /r VSUBPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Subtract packed single-precision floating-point values in xmm3/mem from xmm2 and stores result in xmm1
VEX.NDS.256.0F.WIG 5C /r VSUBPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Subtract packed single-precision floating-point values in ymm3/mem from ymm2 and stores result in ymm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD subtract of the eight or sixteen packed single-precision floating-point values in the second Source operand from the First Source operand, and stores the packed single-precision floating-point results in the destination operand.

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VSUBPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] - SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] - SRC2[127:96]
DEST[159:128] ← SRC1[159:128] - SRC2[159:128]
DEST[191:160]← SRC1[191:160] - SRC2[191:160]
DEST[223:192] ← SRC1[223:192] - SRC2[223:192]
DEST[255:224] ← SRC1[255:224] - SRC2[255:224].

VSUBPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] - SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] - SRC2[127:96]
DEST[255:128] ← 0

SUBPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] - SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] - SRC2[127:96]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VSUBPS __m256 _mm256_sub_ps (__m256 a, __m256 b);

SUBPS __m128 _mm_sub_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

SUBSD- Subtract Scalar Double Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5C /r SUBSD xmm1, xmm2/m64	A	V/V	SSE2	Subtract the low double-precision floating-point value in xmm2/mem from xmm1 and store the result in xmm1
VEX.NDS.LIG.F2.0F.WIG 5C /r VSUBSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Subtract the low double-precision floating-point value in xmm3/mem from xmm2 and store the result in xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

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

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

Operation

VSUBSD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0

SUBSD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] - SRC[63:0]
DEST[255:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

SUBSD __m128d _mm_sub_sd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

SUBSS- Subtract Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5C /r SUBSS xmm1, xmm2/m32	A	V/V	SSE	Subtract the low single-precision floating-point value in xmm2/mem from xmm1 and store the result in xmm1
VEX.NDS.LIG.F3.0F.WIG 5C /r VSUBSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Subtract the low single-precision floating-point value in xmm3/mem from xmm2 and store the result in xmm1

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Subtract the low single-precision floating-point values from the second source operand and the first source operand and store the double-precision floating-point result in the destination operand.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

Operation

VSUBSS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[255:128] ← 0

SUBSS (128-bit Legacy SSE version)

DEST[31:0] ← DEST[31:0] - SRC[31:0]
DEST[255:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

SUBSS __m128 _mm_sub_ss (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

UCOMISD - Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2E /r UCOMISD xmm1, xmm2/m64	A	V/V	SSE2	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2E /r VUCOMISD xmm1, xmm2/m64	A	V/V	AVX	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

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

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory location.

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

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

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

UCOMISD (all versions)

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

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_ucomieq_sd(__m128d a, __m128d b)

int _mm_ucomilt_sd(__m128d a, __m128d b)

int _mm_ucomile_sd(__m128d a, __m128d b)

int _mm_ucomigt_sd(__m128d a, __m128d b)

int _mm_ucomige_sd(__m128d a, __m128d b)

int _mm_ucomineq_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal

Other Exceptions

See Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

UCOMISS - Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 2E /r UCOMISS xmm1, xmm2/m32	A	V/V	SSE	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.0F.WIG 2E /r VUCOMISS xmm1, xmm2/m32	A	V/V	AVX	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

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

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

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

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

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

UCOMISS (all versions)

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

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_ucomieq_ss(__m128 a, __m128 b)

int _mm_ucomilt_ss(__m128 a, __m128 b)

int _mm_ucomile_ss(__m128 a, __m128 b)

int _mm_ucomigt_ss(__m128 a, __m128 b)

int _mm_ucomige_ss(__m128 a, __m128 b)

int _mm_ucomineq_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (if SNaN Operands), Denormal

Other Exceptions

See Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

UNPCKHPD- Unpack and Interleave High Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 15 /r UNPCKHPD xmm1, xmm2/m128	A	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.66.0F.WIG 15 /r VUNPCKHPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.66.0F.WIG 15 /r VUNPCKHPD ymm1,ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the high double-precision floating-point values from the first source operand and the second source operand. See Figure 4-15 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B,.

128-bit versions

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: T second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VUNPCKHPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[191:128]←SRC1[255:192]
DEST[255:192]←SRC2[255:192]

VUNPCKHPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[255:128] ← 0

UNPCKHPD (128-bit Legacy SSE version)

DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPD __m256d _mm256_unpackhi_pd(__m256d a, __m256d b)

UNPCKHPD __m128d _mm_unpackhi_pd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

UNPCKHPS- Unpack and Interleave High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 15 /r UNPCKHPS xmm1, xmm2/m128	A	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.0F.WIG 15 /r VUNPCKHPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.0F.WIG 15 /r VUNPCKHPS ymm1,ymm2,ymm3/m256	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the high single-precision floating-point values from the first source operand and the second source operand.

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced

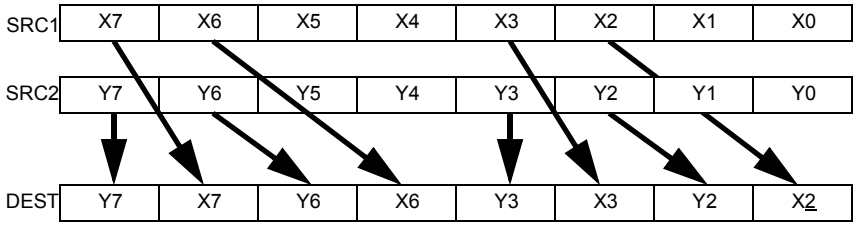


Figure 5-29. VUNPCKHPS Operation

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: T second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VUNPCKHPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[159:128] ← SRC1[223:192]
DEST[191:160] ← SRC2[223:192]
DEST[223:192] ← SRC1[255:224]
DEST[255:224] ← SRC2[255:224]

VUNPCKHPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[255:128] ← 0

UNPCKHPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPS __m256 _mm256_unpackhi_ps (__m256 a, __m256 b);

UNPCKHPS __m128 _mm_unpackhi_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

UNPCKLPD- Unpack and Interleave Low Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 14 /r UNPCKLPD xmm1, xmm2/m128	A	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.66.0F.WIG 14 /r VUNPCKLPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values low high quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.66.0F.WIG 14 /r VUNPCKLPD ymm1,ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values low high quadwords of ymm2 and ymm3/m256.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the low double-precision floating-point values from the first source operand and the second source operand

128-bit versions:
When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: T second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VUNPCKLPD (VEX.256 encoded version)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[191:128] ← SRC1[191:128]
DEST[255:192] ← SRC2[191:128]

VUNPCKLPD (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[255:128] ← 0

UNPCKLPD (128-bit Legacy SSE version)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKLPD __m256d _mm256_unpacklo_pd(__m256d a, __m256d b)

UNPCKLPD __m128d _mm_unpacklo_pd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

UNPCKLPS- Unpack and Interleave Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 14 /r UNPCKLPS xmm1, xmm2/m128	A	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.0F.WIG 14 /r VUNPCKLPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.0F.WIG 14 /r VUNPCKLPS ymm1,ymm2,ymm3/m256	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/m256.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the low single-precision floating-point values from the first source operand and the second source operand

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced

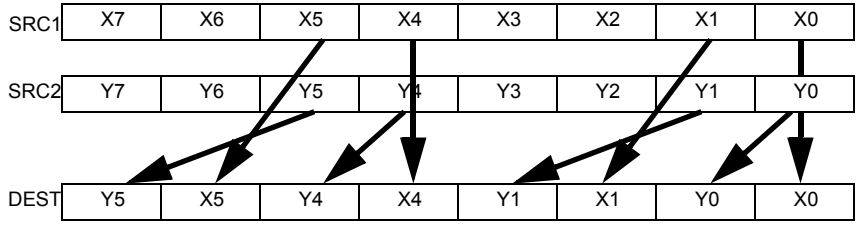


Figure 5-30. VUNPCKLPS Operation

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: T second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

UNPCKLPS (VEX.256 encoded version)
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
DEST[159:128] ← SRC1[159:128]
DEST[191:160] ← SRC2[159:128]
DEST[223:192] ← SRC1[191:160]
DEST[255:224] ← SRC2[191:160]

VUNPCKLPS (VEX.128 encoded version)
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
DEST[255:128] ← 0

UNPCKLPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKLPS __m256 _mm256_unpacklo_ps (__m256 a, __m256 b);

UNPCKLPS __m128 _mm_unpacklo_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

XORPD- Bitwise Logical XOR of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 57/r XORPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical XOR of packed double-precision floating-point values in xmm1 and xmm2/mem
VEX.NDS.128.66.0F.WIG 57 /r VXORPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in xmm2 and xmm3/mem
VEX.NDS.256.66.0F.WIG 57 /r VXORPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical XOR of the two or four packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VXORPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE XOR SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE XOR SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE XOR SRC2[255:192]

VXORPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE XOR SRC2[127:64]
DEST[255:128] ← 0

XORPD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] BITWISE XOR SRC[63:0]
DEST[127:64] ← DEST[127:64] BITWISE XOR SRC[127:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VXORPD __m256d _mm256_xor_pd (__m256d a, __m256d b);

XORPD __m128d _mm_xor_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

XORPS- Bitwise Logical XOR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 57 /r XORPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical XOR of packed single-precision floating-point values in xmm1 and xmm2/mem
VEX.NDS.128.0F.WIG 57 /r VXORPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/mem
VEX.NDS.256.0F.WIG 57 /r VXORPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/mem

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical XOR of the four or eight packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VXORPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[159:128] ← SRC1[159:128] BITWISE XOR SRC2[159:128]
DEST[191:160]← SRC1[191:160] BITWISE XOR SRC2[191:160]
DEST[223:192] ← SRC1[223:192] BITWISE XOR SRC2[223:192]
DEST[255:224] ← SRC1[255:224] BITWISE XOR SRC2[255:224].

VXORPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[255:128] ← 0

XORPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VXORPS __m256 _mm256_xor_ps (__m256 a, __m256 b);

XORPS __m128 _mm_xor_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

VZEROALL- Zero All YMM registers

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.0F.W0 77 VZEROALL	A	V/V	AVX	Zero all YMM registers

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

Description

The instruction zeros contents of all XMM or YMM registers.

Note: VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD. VEX.W must be 0, A #UD will occur otherwise.

In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

Operation

VZEROALL (VEX.256 encoded version)

```
IF (64-bit mode)
    YMM0[255:0] ← 0
    YMM1[255:0] ← 0
    YMM2[255:0] ← 0
    YMM3[255:0] ← 0
    YMM4[255:0] ← 0
    YMM5[255:0] ← 0
    YMM6[255:0] ← 0
    YMM7[255:0] ← 0
    YMM8[255:0] ← 0
    YMM9[255:0] ← 0
    YMM10[255:0] ← 0
    YMM11[255:0] ← 0
    YMM12[255:0] ← 0
    YMM13[255:0] ← 0
    YMM14[255:0] ← 0
    YMM15[255:0] ← 0
ELSE
    YMM0[255:0] ← 0
```

```
    YMM1[255:0] ← 0
    YMM2[255:0] ← 0
    YMM3[255:0] ← 0
    YMM4[255:0] ← 0
    YMM5[255:0] ← 0
    YMM6[255:0] ← 0
    YMM7[255:0] ← 0
    YMM8-15: Unmodified
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VZEROALL _mm256_zeroall()

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 8

#UD	If VEX.W=1

VZEROUPPER- Zero Upper bits of YMM registers

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.0F 77 VZEROUPPER	A	V/V	AVX	Zero upper 128 bits of all YMM registers

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

Description

The instruction zeros the upper 128 bits of all YMM registers. The lower 128-bits of the registers (the corresponding XMM registers) are unmodified.

This instruction is recommended when transitioning between AVX and legacy SSE code - it will eliminate performance penalties caused by false dependencies.

Note: VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. VEX.W must be 0, A #UD will occur otherwise.

In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

Operation

VZEROUPPER

```
IF (64-bit mode)
    YMM0[255:128] ← 0
    YMM1[255:128] ← 0
    YMM2[255:128] ← 0
    YMM3[255:128] ← 0
    YMM4[255:128] ← 0
    YMM5[255:128] ← 0
    YMM6[255:128] ← 0
    YMM7[255:128] ← 0
    YMM8[255:128] ← 0
    YMM9[255:128] ← 0
    YMM10[255:128] ← 0
    YMM11[255:128] ← 0
    YMM12[255:128] ← 0
    YMM13[255:128] ← 0
    YMM14[255:128] ← 0
```

```
    YMM15[255:128] ← 0
ELSE
    YMM0[255:128] ← 0
    YMM1[255:128] ← 0
    YMM2[255:128] ← 0
    YMM3[255:128] ← 0
    YMM4[255:128] ← 0
    YMM5[255:128] ← 0
    YMM6[255:128] ← 0
    YMM7[255:128] ← 0
    YMM8-15: unmodified
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VZEROUPPER _mm256_zeroupper()

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 8

#UD	If VEX.W=1

XSAVEOPT—Save Processor Extended States Optimized

Opcode	Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F AE /6	XSAVEOPT *mem*	A	V/V	XSAVEOPT	Save processor extended states specified in EDX:EAX to memory, optimizing the state save operation if possible.
REX.W + 0F AE /6	XSAVEOPT64 *mem*	A	V/V	XSAVEOPT	Save processor extended states specified in EDX:EAX to memory, optimizing the state save operation if possible.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (w)	NA	NA	NA

Description

Performs a full or partial save of the enabled processor state components to a memory address specified in the destination operand. A full or partial save of the processor states is specified by an implicit mask operand via the register pair, EDX:EAX. The destination operand is a memory location that must be 64-byte aligned. The hardware may optimize the manner in which data is saved. The performance of this instruction will be equal or better than using the XSAVE instruction.

The implicit 64-bit mask operand in EDX:EAX specifies the subset of enabled processor state components to save into the XSAVE/XRSTOR save area. The XSAVE/XRSTOR save area comprises of individual save area for each processor state components and a header section, see Table 3-3.

The bit assignment used for the EDX:EAX register pair matches the XFEATURE_ENABLED_MASK register. For the XSAVEOPT instruction, software can specify "1" in any bit position of EDX:EAX, irrespective of whether the corresponding bit position in XFEATURE_ENABLED_MASK is valid for the processor. The bit vector in EDX:EAX is "anded" with the XFEATURE_ENABLED_MASK to determine which save area will be written.

The content layout of the XSAVE/XRSTOR save area is architecturally defined to be extendable and enumerated via the sub-leaves of CPUID.0DH leaf. The extendable framework of the XSAVE/XRSTOR layout is depicted by Table 3-3. The layout of the XSAVE/XRSTOR save area is fixed and may contain non-contiguous individual save areas. The XSAVE/XRSTOR save area is not compacted if some features are not saved or are not supported by the processor and/or by system software.

The layout of the register fields of first 512 bytes of the XSAVE/XRSTOR is the same as the FXSAVE/FXRSTOR area. But XSAVE/XRSTOR organizes the 512 byte area as x87 FPU states (including FPU operation states, x87/MMX data registers), MXCSR (including MXCSR_MASK), and XMM registers.

The processor writes 1 or 0 to each.HEADER.XSTATE_BV[i] bit field of an enabled processor state component in a manner that is consistent to XRSTOR's interaction with HEADER.XSTATE_BV.

The state updated to the XSAVE/XRSTOR area may be optimized as follows:

- If the state is in its initialized form, the corresponding XSTATE_BV bit may be set to 0, and the corresponding processor state component that is indicated as initialized will not be saved to memory.

A processor state component save area is not updated if either one of the corresponding bits in the mask operand or the XFEATURE_ENABLED_MASK register is 0. The processor state component that is updated to the save area is computed by bitwise AND of the mask operand (EDX:EAX) with XFEATURE_ENABLED_MASK.

HEADER.XSTATE_BV is updated to reflect the data that is actually written to the save area. A "1" bit in the header indicates the contents of the save area corresponding to that bit are valid. A "0" bit in the header indicates that the state corresponding to that bit is in its initialized form. The memory image corresponding to a "0" bit may or may not contain the correct (initialized) value since only the header bit (and not the save area contents) is updated when the header bit value is 0. XRSTOR will ensure the correct value is placed in the register state regardless of the value of the save area when the header bit is zero.

Operation

```
TMP_MASK[62:0]    (EDX[30:0] << 32 ) OR EAX[31:0] ) AND XFEATURE_ENABLE_MASK[62:0];
FOR i = 0, 62 STEP 1
    IF (TMP_MASK[i] = 1)
    THEN
        If not HW_CAN_OPTIMIZE_SAVE
        THEN
            CASE ( i ) of
                0: DEST.FPUSSESAVE_Area[x87 FPU]   processor state[x87 FPU];
                1: DEST.FPUSSESAVE_Area[SSE]   processor state[SSE];
                    // SSE state include MXCSR
                2: DEST.EXT_SAVE_Area2[YMM]   processor state[YMM];
                DEFAULT: // i corresponds to a valid sub-leaf index of CPUID leaf 0DH
                    DEST.Ext_Save_Area[ i ]   processor state[i] ;
            ESAC:
        FI;
        DEST.HEADER.XSTATE_BV[i]   INIT_FUNCTION[i];
    FI;
NEXT;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CPUID.(EAX=0DH, ECX=01H):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CPUID.(EAX=0DH, ECX=01H):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CPUID.(EAX=0DH, ECX=01H):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H prefix is used.

This page was intentionally left blank.

CHAPTER 6
INSTRUCTION SET REFERENCE - FMA

6.1 FMA INSTRUCTION SET REFERENCE

This section describes FMA instructions in details. Conventions and notations of instruction format can be found in Section 5.1.

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

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 98 /r VFMADD132PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 A8 /r VFMADD213PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 B8 /r VFMADD231PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W1 98 /r VFMADD132PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, add to ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W1 A8 /r VFMADD213PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, add to ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W1 B8 /r VFMADD231PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, add to ymm0 and put result in ymm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a set of SIMD multiply-add computation on packed double-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADD213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADD231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

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

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

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
```

```
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] + SRC2[n+63:n])
}
IF (VEX.128) THEN
DEST[255:128] ← 0
FI
```

VFMADD213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] + SRC3[n+63:n])
}
IF (VEX.128) THEN
DEST[255:128] ← 0
FI
```

VFMADD231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] + DEST[n+63:n])
}
IF (VEX.128) THEN
DEST[255:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132PD __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);

VFMADD213PD __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);

VFMADD231PD __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);

VFMADD132PD __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

VFMADD213PD __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

VFMADD231PD __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

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

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 98 /r VFMADD132PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 A8 /r VFMADD213PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 B8 /r VFMADD231PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 98 /r VFMADD132PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, add to ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 A8 /r VFMADD213PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, add to ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.0 B8 /r VFMADD231PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, add to ymm0 and put result in ymm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a set of SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision

floating-point values in the third source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADD213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADD231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

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

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

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior"

Operation

In the operations below, "+" and "*" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
```

```
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

VFMADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

VFMADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132PS __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);

VFMADD213PS __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);

VFMADD231PS __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);

VFMADD132PS __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

VFMADD213PS __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

VFMADD231PS __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

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

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 99 /r VFMADD132SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 A9 /r VFMADD213SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm1, add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 B9 /r VFMADD231SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

VFMADD132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMADD213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMADD231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third

source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior"

Operation

In the operations below, "+" and "*" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMADD132SD DEST, SRC2, SRC3

DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[255:128] ← 0

VFMADD213SD DEST, SRC2, SRC3

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[255:128] ← 0

VFMADD231SD DEST, SRC2, SRC3

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132SD __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

VFMADD213SD __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

VFMADD231SD __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

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

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 99 /r VFMADD132SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 A9 /r VFMADD213SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm1, add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 B9 /r VFMADD231SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in

the third source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior"

Operation

In the operations below, "+" and "*" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMADD132SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[255:128] ← 0

VFMADD213SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[255:128] ← 0

VFMADD231SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[63:0] + DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132SS __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);

VFMADD213SS __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);

VFMADD231SS __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

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

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, add/subtract elements in xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, add/subtract elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add/subtract elements in xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, add/subtract elements in ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, add/subtract elements in ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, add/subtract elements in ymm0 and put result in ymm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFMADDSUB132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

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

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

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMADDSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[255:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] - SRC2[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] + SRC2[255:192]
FI
```

VFMADDSUB213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
    DEST[255:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] - SRC3[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] + SRC3[255:192]
FI
```

VFMADDSUB231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[255:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] - DEST[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] + DEST[255:192]
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUB132PD __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);

VFMADDSUB213PD __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);

VFMADDSUB231PD __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);

VFMADDSUB132PD __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

VFMADDSUB213PD __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

VFMADDSUB231PD __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

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

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, add/subtract elements in xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 A6 /r VFMADDSUB213PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, add/subtract elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 B6 /r VFMADDSUB231PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, add/subtract elements in xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, add/subtract elements in ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, add/subtract elements in ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, add/subtract elements in ymm0 and put result in ymm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFMADDSUB132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

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

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

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMADDSUB132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1 {
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] +
SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

VFMADDSUB213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1 {
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] +
SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

VFMADDSUB231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1 {
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
```

```
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] +
DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUB132PS __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);

VFMADDSUB213PS __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);

VFMADDSUB231PS __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);

VFMADDSUB132PS __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

VFMADDSUB213PS __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

VFMADDSUB231PS __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

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

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, subtract/add elements in xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, subtract/add elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, subtract/add elements in xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, subtract/add elements in ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, subtract/add elements in ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, subtract/add elements in ymm0 and put result in ymm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFMSUBADD132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

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

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

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMSUBADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[255:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] + SRC2[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] - SRC2[255:192]
FI
```

VFMSUBADD213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
    DEST[255:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] + SRC3[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] - SRC3[255:192]
FI
```

VFMSUBADD231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[255:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] + DEST[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] - DEST[255:192]
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADD132PD __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);

VFMSUBADD213PD __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);

VFMSUBADD231PD __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);

VFMSUBADD132PD __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

VFMSUBADD213PD __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

VFMSUBADD231PD __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

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

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, subtract/add elements in xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 A7 /r VFMSUBADD213PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, subtract/add elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 B7 /r VFMSUBADD231PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, subtract/add elements in xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, subtract/add elements in ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, subtract/add elements in ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, subtract/add elements in ymm0 and put result in ymm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFMSUBADD132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

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

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

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMSUBADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1 {
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] -
SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

VFMSUBADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1 {
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] +SRC3[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] -
SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

VFMSUBADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1 {
    n = 64*i;
```

```
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] -
DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADD132PS __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);

VFMSUBADD213PS __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);

VFMSUBADD231PS __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);

VFMSUBADD132PS __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

VFMSUBADD213PS __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

VFMSUBADD231PS __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

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

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9A /r VFMSUB132PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 AA /r VFMSUB213PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 BA /r VFMSUB231PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W1 9A /r VFMSUB132PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, subtract ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W1 AA /r VFMSUB213PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, subtract ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W1 BA /r VFMSUB231PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, subtract ymm0 and put result in ymm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

VFMSUB132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

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

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior"

Operation

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

VFMSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
```

```
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] - SRC2[n+63:n])
}
IF (VEX.128) THEN
DEST[255:128] ← 0
FI
```

VFMSUB213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] - SRC3[n+63:n])
}
IF (VEX.128) THEN
DEST[255:128] ← 0
FI
```

VFMSUB231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] - DEST[n+63:n])
}
IF (VEX.128) THEN
DEST[255:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132PD __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);

VFMSUB213PD __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);

VFMSUB231PD __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);

VFMSUB132PD __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

VFMSUB213PD __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

VFMSUB231PD __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

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

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9A /r VFMSUB132PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 AA /r VFMSUB213PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 BA /r VFMSUB231PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 9A /r VFMSUB132PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, subtract ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 AA /r VFMSUB213PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, subtract ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.0 BA /r VFMSUB231PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, subtract ymm0 and put result in ymm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a set of SIMD multiply-subtract computation on packed single-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB231PS: Multiplies the four or eight packed single-precision floating-point values from the second source to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

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

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

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior".

Operation

In the operations below, "+" and "*" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMSUB132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
```

```
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

VFMSUB213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

VFMSUB231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132PS __m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);

VFMSUB213PS __m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);

VFMSUB231PS __m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);

VFMSUB132PS __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

VFMSUB213PS __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

VFMSUB231PS __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

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

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9B /r VFMSUB132SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 AB /r VFMSUB213SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm1, subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 BB /r VFMSUB231SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

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

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

VFMSUB231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third

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

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior".

Operation

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

VFMSUB132SD DEST, SRC2, SRC3

DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[255:128] ← 0

VFMSUB213SD DEST, SRC2, SRC3

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[255:128] ← 0

VFMSUB231SD DEST, SRC2, SRC3

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132SD __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

VFMSUB213SD __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

VFMSUB231SD __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

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

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9B /r VFMSUB132SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 AB /r VFMSUB213SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm1, subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 BB /r VFMSUB231SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

VFMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the the low packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third

source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior"

Operation

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

VFMSUB132SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[255:128] ← 0

VFMSUB213SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[255:128] ← 0

VFMSUB231SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[63:0] - DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132SS __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

VFMSUB213SS __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

VFMSUB231SS __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMADD132PD/VFNMADD213PD/VFNMADD231PD - Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9C /r VFNMADD132PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 AC /r VFNMADD213PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, negate the multiplication result and add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 BC /r VFNMADD231PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and add to xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W1 9C /r VFNMADD132PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and add to ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W1 AC /r VFNMADD213PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, negate the multiplication result and add to ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W1 BC /r VFNMADD231PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and add to ymm0 and put result in ymm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMADD132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision

floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

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

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

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”.

Operation

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

VFNMADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
```

```
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(DEST[n+63:n]*SRC3[n+63:n]) + SRC2[n+63:n])
}
IF (VEX.128) THEN
DEST[255:128] ← 0
FI
```

VFNMADD213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*DEST[n+63:n]) + SRC3[n+63:n])
}
IF (VEX.128) THEN
DEST[255:128] ← 0
FI
```

VFNMADD231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*SRC3[n+63:n]) + DEST[n+63:n])
}
IF (VEX.128) THEN
DEST[255:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADD132PD __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);

VFNMADD213PD __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);

VFNMADD231PD __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);

VFNMADD132PD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

VFNMADD213PD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

VFNMADD231PD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMADD132PS/VFNMADD213PS/VFNMADD231PS - Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9C /r VFNMADD132PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 AC /r VFNMADD213PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, negate the multiplication result and add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 BC /r VFNMADD231PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and add to xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 9C /r VFNMADD132PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and add to ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 AC /r VFNMADD213PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, negate the multiplication result and add to ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.0 BC /r VFNMADD231PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and add to ymm0 and put result in ymm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMADD132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision

floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

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

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

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFNMADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
```

```
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (DEST[n+31:n]*SRC3[n+31:n]) + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

VFNMADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

VFNMADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (SRC2[n+31:n]*SRC3[n+31:n]) + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADD132PS __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);

VFNMADD213PS __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);

VFNMADD231PS __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);

VFNMADD132PS __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

VFNMADD213PS __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

VFNMADD231PS __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMADD132SD/VFNMADD213SD/VFNMADD231SD - Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9D /r VFNMADD132SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm2/mem, negate the multiplication result and add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 AD /r VFNMADD213SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm1, negate the multiplication result and add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 BD /r VFNMADD231SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2/mem, negate the multiplication result and add to xmm0 and put result in xmm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMADD132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMADD213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMADD231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand,

performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior".

.Operation

In the operations below, "+" and "*" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFNMADD132SD DEST, SRC2, SRC3

DEST[63:0] ← RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[255:128] ← 0

VFNMADD213SD DEST, SRC2, SRC3

DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[255:128] ← 0

VFNMADD231SD DEST, SRC2, SRC3

DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADD132SD __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);

VFNMADD213SD __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);

VFNMADD231SD __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMADD132SS/VFNMADD213SS/VFNMADD231SS - Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9D /r VFNMADD132SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm2/mem, negate the multiplication result and add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 AD /r VFNMADD213SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm1, negate the multiplication result and add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 BD /r VFNMADD231SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2/mem, negate the multiplication result and add to xmm0 and put result in xmm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand,

performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior".

Operation

In the operations below, "+" and "*" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFNMADD132SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[255:128] ← 0

VFNMADD213SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[255:128] ← 0

VFNMADD231SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[63:0]) + DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADD132SS __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);

VFNMADD213SS __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);

VFNMADD231SS __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD - Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9E /r VFNMSUB132PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 AE /r VFNMSUB213PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, negate the multiplication result and subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 BE /r VFNMSUB231PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W1 9E /r VFNMSUB132PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and subtract ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W1 AE /r VFNMSUB213PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, negate the multiplication result and subtract ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W1 BE /r VFNMSUB231PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and subtract ymm0 and put result in ymm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMSUB132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

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

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic

Behavior”.

Operation

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

VFNMSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (DEST[n+63:n]*SRC3[n+63:n]) - SRC2[n+63:n])
}
IF (VEX.128) THEN
DEST[255:128] ← 0
FI
```

VFNMSUB213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*DEST[n+63:n]) - SRC3[n+63:n])
}
IF (VEX.128) THEN
DEST[255:128] ← 0
FI
```

VFNMSUB231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*SRC3[n+63:n]) - DEST[n+63:n])
}
```

```
IF (VEX.128) THEN
DEST[255:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132PD __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);

VFNMSUB213PD __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);

VFNMSUB231PD __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);

VFNMSUB132PD __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

VFNMSUB213PD __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

VFNMSUB231PD __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS - Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9E /r VFNMSUB132PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 AE /r VFNMSUB213PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, negate the multiplication result and subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 BE /r VFNMSUB231PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 9E /r VFNMSUB132PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and subtract ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 AE /r VFNMSUB213PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, negate the multiplication result and subtract ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.0 BE /r VFNMSUB231PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and subtract ymm0 and put result in ymm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

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

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

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

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

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFNMSUB132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

VFNMSUB213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI
```

VFNMSUB231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
```

FI

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132PS __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);

VFNMSUB213PS __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);

VFNMSUB231PS __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);

VFNMSUB132PS __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

VFNMSUB213PS __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

VFNMSUB231PS __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD - Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9F /r VFNMSUB132SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 AF /r VFNMSUB213SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm1, negate the multiplication result and subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 BF /r VFNMSUB231SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0 and put result in xmm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMSUB132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts

the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior".

Operation

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

VFNMSUB132SD DEST, SRC2, SRC3

DEST[63:0] ← RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[255:128] ← 0

VFNMSUB213SD DEST, SRC2, SRC3

DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[255:128] ← 0

VFNMSUB231SD DEST, SRC2, SRC3

DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132SD __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);

VFNMSUB213SD __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);

VFNMSUB231SD __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS - Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op En	Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9F /r VFNMSUB132SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 AF /r VFNMSUB213SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm1, negate the multiplication result and subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 BF /r VFNMSUB231SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0 and put result in xmm0.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-

precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior".

Operation

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

VFNMSUB132SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[255:128] ← 0

VFNMSUB213SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[255:128] ← 0

VFNMSUB231SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[63:0]) - DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132SS __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);

VFNMSUB213SS __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);

VFNMSUB231SS __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

This page was intentionally left blank.

CHAPTER 7
POST-32NM PROCESSOR INSTRUCTIONS

7.1 OVERVIEW

This chapter describes additional instructions targeted for Intel 64 architecture processors in process technology smaller than 32 nm. These instructions include

- Two instructions are added to support 16-bit floating-point data type conversion to and from single-precision floating-point type. Conversion to packed 16-bit floating-point values from packed single-precision floating-point values also provides rounding control using an immediate byte. These float-16 instructions converts packed data types of different sizes following the same manner as the 256-bit vector SIMD extension, AVX.
- One instruction that generates random numbers of 16/32/64 bit wide random integers. The random number generator instruction operates on general-purpose registers.
- Four instructions that allow software working in 64-bit environment to read and write FS base and GS base registers in all privileged levels.

7.2 CPUID DETECTION OF NEW INSTRUCTIONS

Application using float 16 instruction must follow a detection sequence similar to AVX to ensure:

- The OS has enabled YMM state management support,
- The processor support AVX as indicated by the CPUID feature flag, i.e. CPUID.01H:ECX.AVX[bit 28] = 1.
- The processor support 16-bit floating-point conversion instructions via a CPUID feature flag (CPUID.01H:ECX.F16C[bit 29] = 1).

Application detection of Float-16 conversion instructions follow the general procedural flow in Figure 7-1.

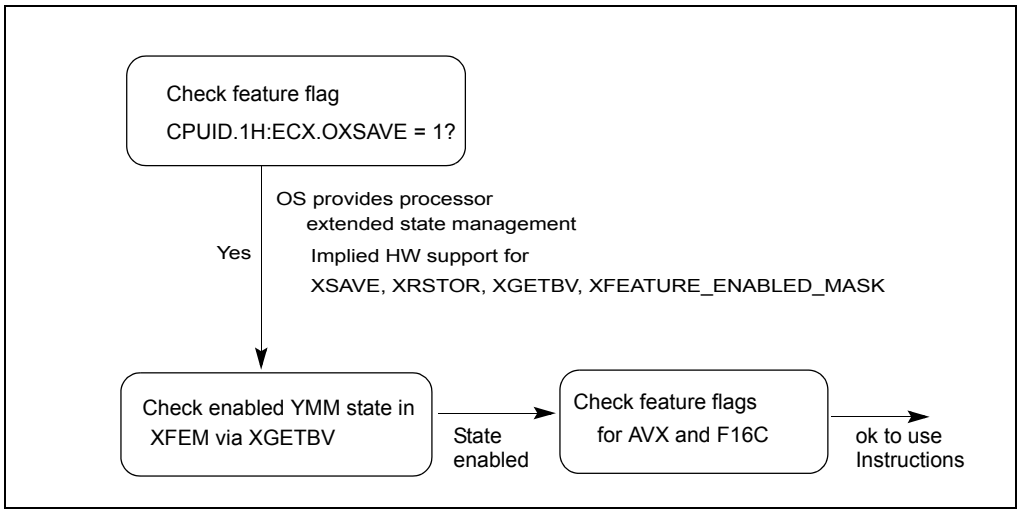


Figure 7-1. General Procedural Flow of Application Detection of Float-16

```
---------------------------------------------------------------------------------------
INT supports_f16c()
{       ; result in eax
        mov eax, 1
        cpuid
        and ecx, 038000000H
        cmp ecx, 038000000H; check OSXSAVE, AVX, F16C feature flags
         jne not_supported
        ; processor supports AVX,F16C instructions and XGETBV is enabled by OS
        mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
        XGETBV; result in EDX:EAX
        and eax, 06H
        cmp eax, 06H; check OS has enabled both XMM and YMM state support
        jne not_supported
        mov eax, 1
        jmp done
        NOT_SUPPORTED:
        mov eax, 0
        done:
```

```
}
--------------------------------------------------------------------------------
```

RDRND, RDFSBASE/RDGSBASE, WRFSBASE/WRGSBASE operates on general purpose registers only.

Before software attempts to use the RDRND instruction, it must check that the CPUID feature flag indicating processor supports RDRND, i.e.if CPUID.01H:ECX.RDRND[bit 30] = 1.

CPUID.(EAX=07H, ECX=0H):EBX.FGSBASE[bit 0] = 1 indicates availability of instructions that are primarily targeting use by system software manipulating the base of FS and GS segments. These instructions require enabling by OS (see Section 7.5). An OS may provide programming interfaces indicating its support of application use of RDFSBASE/RDGSBASE, WRFSBASE/WRGSBASE instructions.

7.3 16-BIT FLOATING-POINT DATA TYPE SUPPORT

Two new instructions support half-precision floating-point data type with conversion to and from single-precision floating-point data types. Table 7-1 gives the length, precision, and approximate normalized range that can be represented by half and single precision data types. Denormal values are also supported in these types.

Table 7-1. Length, Precision, and Range of Floating-Point Data Types

Data Type	Length	Precision (Bits)	Approximate Normalized Range	
			Binary	Decimal
Half Precision	16	11	2^{-14} to 2^{15}	3.1×10^{-5} to 6.50×10^{4}
Single Precision	32	24	2^{-126} to 2^{127}	1.18×10^{-38} to 3.40×10^{38}
Double Precision	64	53	2^{-1022} to 2^{1023}	2.23×10^{-308} to 1.79×10^{308}

Table 7-2 shows the floating-point encodings for zeros, denormalized finite numbers, normalized finite numbers, infinities, and NaNs for each of the three floating-point data types.

Table 7-2. Half-Precision Floating-Point Number and NaN Encodings

Class		Sign	Biased Exponent	Significand	
				Integer[1]	Fraction
Positive	+∞	0	11..11	1	00..00
	+Normals	0 . . 0	11..10 . . 00..01	1 . . 1	11..11 . . 00..00
	+Denormals	0 . . 0	00..00 . . 00..00	0 . . 0	11.11 . . 00..01
	+Zero	0	00..00	0	00..00
Negative	–Zero	1	00..00	0	00..00
	–Denormals	1 . . 1	00..00 . . 00..00	0 . . 0	00..01 . . 11..11
	–Normals	1 . . 1	00..01 . . 11..10	1 . . 1	00..00 . . 11..11
	-▪	1	11..11	1	00..00
NaNs	SNaN	X	11..11	1	0X..XX[2]
	QNaN	X	11..11	1	1X..XX
	QNaN Floating-Point Indefinite	1	11..11	1	10..00
	Half-Precision: Single-Precision; Double-Precision;		← 5Bits → ← 8 Bits → ← 11 Bits →		← 10 Bits → ← 23Bits → ← 52Bits →

NOTES:

1. Integer bit is implied and not stored in memory format.
2. The fraction for SNaN encodings must be non-zero with the most-significant bit 0.
3. The most-significant bit of the fraction for QNaN encoding must be 1.

Half-precision floating-point data type consists of a sign bit, a 5-bit exponent field, and a 11-bit significand field. The 11-bit significand consists of an implied integer bit that is not stored and a 10-bit fraction field that is stored as the 10 least-significand bits along with the sign bit (bit 15) and the exponent field (bits 14:10), see Figure 7-2.

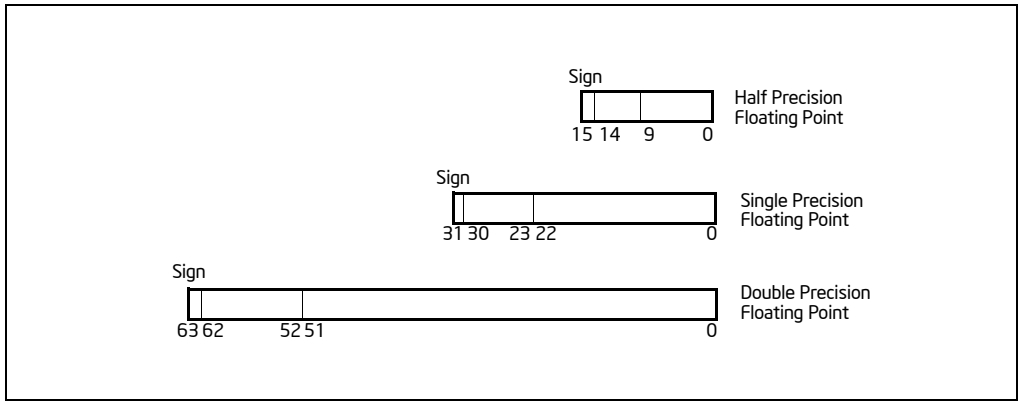


Figure 7-2. Floating-Point Data Types

The exponent of each floating-point data type is encoded in biased format, the bias constant is 15 for half-precision floating-point data type. When storing floating-point values in memory, half-precision values are stored in 2 consecutive bytes in memory. Table 7-3 shows how the real number 178.125 (in ordinary decimal format) is stored in IEEE Standard 754 floating-point format. The table lists a progression of real number notations that leads to the half-precision, 16-bit floating-point format.

Table 7-3. Real and Floating-Point Number Notation

Notation	Value		
Ordinary Decimal	178.125		
Scientific Decimal	$1.78125E_{10}2$		
Scientific Binary	$1.0110010001E_{2}111$		
Scientific Binary (Biased Exponent)	$1.0110010001E_{2}10110$		
Half-Precision Format	Sign	Biased Exponent	Normalized Significand
	0	10110	0110010001 1. (Implied)

7.3.1 Half-Precision Floating-Point Conversion

Half-precision floating-point values are not used by the processor directly for arithmetic operations. Two instructions, VCVTPH2PS, VCVTPS2PH, provide conversion between half-precision and single-precision floating-point values.

The conversion operations of VCVTPS2PH allow programmer to specify rounding control using control fields in an immediate byte. The effect of the immediate byte are listed in Table 7-4.

Rounding control can use Imm[2] to select an override RC field specified in Imm[1:0] or use MXCSR setting.

Table 7-4. Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions

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

VCVTPH2PS and VCVTPS2PH are subject to SIMD floating-point exceptions. Specifically, they can cause invalid operation exception (see Table 7-6), the result of which is shown in Table 7-5.

Table 7-5. Non-Numerical Behavior for VCVTPH2PS, VCVTPS2PH

Source Operands	Masked Result	Unmasked Result
QNaN	QNaN1[1]	QNaN1[1] (not an exception)
SNaN	QNaN1[2]	None

NOTES:

1. The half precision output QNaN1 is created from the single precision input QNaN as follows: the sign bit is preserved, the 8-bit exponent FFH is replaced by the 5-bit exponent 1FH, and the 24-bit significand is truncated to an 11-bit significand by removing its 14 least significant bits.

2. The half precision output QNaN1 is created from the single precision input SNaN as follows: the sign bit is preserved, the 8-bit exponent FFH is replaced by the 5-bit exponent 1FH, and the 24-bit significand is truncated to an 11-bit significand by removing its 14 least significant bits. The second most significant bit of the significand is changed from 0 to 1 to convert the signaling NaN into a quiet NaN.

Table 7-6. Invalid Operation for VCVTPH2PS, VCVTPS2PH

Instruction	Condition	Masked Result	Unmasked Result
VCVTPH2PS	SRC = NaN	See Table 7-5	#I=1
VCVTPS2PH	SRC = NaN	SeeTable 7-5	#I=1

VCVTPS2PH can cause denormal exceptions if the value of the source operand is denormal relative to the numerical range represented by the source format (see Table 7-7).

Table 7-7. Denormal Condition for VCVTPS2PH

Instruction	Condition	Masked Result[1]	Unmasked Result
VCVTPH2PS	SRC is denormal relative to input format[1]	res = Result rounded to the destination precision and using the bounded exponent, but only if no unmasked post-computation exception occurs. #DE unchanged	Same as masked result.
VCVTPS2PH	SRC is denormal relative to input format[1]	res = Result rounded to the destination precision and using the bounded exponent, but only if no unmasked post-computation exception occurs. #DE=1	#DE=1

NOTES:

1. Masked and unmasked result is shown in Table 7-7.

VCVTPS2PH can cause an underflow exception if the result of the conversion is less than the underflow threshold for half-precision floating-point data type , i.e. $| x | < 1.0 * 2^{-14}$.

Table 7-8. Underflow Condition for VCVTPS2PH

Instruction	Condition	Masked Result[1]	Unmasked Result
VCVTPS2PH	Result < smallest destination precision finial normal value[2]	Result = +0 or -0, denormal, normal. #UE =1. #PE = 1 if the result is inexact.	#UE=1, #PE = 1 if the result is inexact.

NOTES:

1. Masked and unmasked result is shown in Table 7-7.
2. If FTZ is not set (MXCSR.FTZ = 1), masked and unmasked result is shown in Table 7-8. If FTZ is set (MXCSR.FTZ = 1), inexact result = +0 or - 0, #PE and #UE are reported.

VCVTPS2PH can cause an overflow exception if the result of the conversion is less than the underflow threshold for half-precision floating-point data type, i.e. $| x | \geq 1.0 * 2^{16}$.

Table 7-9. Overflow Condition for VCVTPS2PH

Instruction	Condition	Masked Result	Unmasked Result
VCVTPS2PH	Result ≥ largest destination precision finial normal value[1]	Result = +Inf or -Inf. #OE=1.	#OE=1.

VCVTPS2PH can cause an inexact exception if the result of the conversion is not exactly representable in the destination format.

Table 7-10. Inexact Condition for VCVTPS2PH

Instruction	Condition	Masked Result[1]	Unmasked Result
VCVTPS2PH	The result is not representable in the destination format	res = Result rounded to the destination precision and using the bounded exponent, but only if no unmasked underflow or overflow conditions occur (this exception can occur in the presence of a masked underflow or overflow). #PE=1.	Only if no underflow/overflow condition occurred, or if the corresponding exceptions are masked: ▪ Set #OE if masked overflow and set result as described above for masked overflow. ▪ Set #UE if masked underflow and set result as described above for masked underflow. If neither underflow nor overflow, result equals the result rounded to the destination precision and using the bounded exponent set #PE = 1.

NOTES:

1. If a source is denormal relative to input format with DM masked and at least one of PM or UM unmasked, then an exception will be raised with DE, UE and PE set.

7.4 VECTOR INSTRUCTION EXCEPTION SPECIFICATION

The exception behavior of instructions operating on YMM states follows the updated classification table of Table 7-11. The instructions VCVTPS2PH and VCVTPS2PH are described by type 11.

Table 7-11. Exception class description

Exception Class	NI Family	Mem arg	Floating Point Exceptions (#XM)
Type 1	AVX, Legacy SSE	16/32 byte explicitly aligned	none
Type 2	AVX, FMA, Legacy SSE	16/32 byte; not explicitly aligned with VEX prefix; explicitly aligned without VEX	yes
Type 3	AVX, FMA,, Legacy SSE	< 16 byte	yes
Type 4	AVX, Legacy SSE	16/32 byte not explicitly aligned with VEX prefix; explicitly aligned without VEX	no
Type 5	AVX, Legacy SSE	< 16 byte	no
Type 6	AVX (no Legacy SSE)	Varies	(At present, none do)
Type 7	AVX, Legacy SSE	none	none
Type 8	AVX	none	none
Type 9	AVX	4 byte	none
Type 10	AVX, Legacy SSE	16/32 byte; not explicitly aligned	no
Type 11	AVX	Not explicitly aligned, no AC#	yes

7.4.1 Exception Type 11 (VEX-only, mem arg no AC, floating-point exceptions)

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix
			X	X	VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1

7.5 FS/GS BASE SUPPORT FOR 64-BIT SOFTWARE

64-bit code can use new instructions to access and modify FS and GS base. These new instructions are available to software in all privilege levels. CR4 register bit 16 allows system software to control the availability of these instructions to software.

CR4.FGSBASE

FGSBASE-Enable Bit (bit 16 of CR4) — Enables RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE instructions in all privilege levels when set. When clear, RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE instructions cause #UD in all privilege level. The default value of this bit is zero after RESET.

RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE instructions are available only 64-bit sub-mode of the IA-32e mode. Access to CR4.FGSBASE is available in all operating modes if CPUID.(EAX=07H, ECX=0H):EBX.FGSBASE is 1.

NOTE

It is highly recommended that REX.W prefix is used with these instructions to read/write full 64-bit value. If REX.W prefix is omitted, when reading from segment base, upper 32-bits will be ignored and will be set to zero in destination registers. If REX.W prefix is omitted for write to segment base, the upper 32-bits of source resister will be ignored and the corresponding bits for segment base will be set to zero. Additionally, if OS enables these instructions, they also context switch GS and FS base to ensure that any changes made by the applications to the segment base are appropriately context switched.

7.6 INSTRUCTION REFERENCE

Conventions and notations of instruction format can be found in Section 5.1.

RDFSBASE/RDGSBASE—Read FS/GS Segment Base Register

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F AE /0 RDFSBASE r32	A	V/I	FSGSBASE	Read FS base register and place the 32-bit result in the destination register.
REX.W + F3 0F AE /0 RDFSBASE r64	A	V/I	FSGSBASE	Read FS base register and place the 64-bit result in the destination register.
F3 0F AE /1 RDGSBASE *r32*	A	V/I	FSGSBASE	Read GS base register and place the 32-bit result in destination register.
REX.W + F3 0F AE /1 RDGSBASE *r64*	A	V/I	FSGSBASE	Read GS base register and place the 64-bit result in destination register.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (w)	NA	NA	NA

Description

Loads the FS or GS segment base register into a general purpose register indicated by the modR/M:r/m field. The destination operand is a 32 or 64-bit general purpose register. The REX.W prefix indicates the operand size is 64 bit. If no REX.W prefix is used then the operand size is 32 bit and the upper 32 bits of the FS or GS base are ignored and bits [63:32] of the destination register will be written to 0.

This instruction is supported only in 64-bit sub mode of the IA-32e mode.

Operation

```
If OperandSize = 64 then
    DEST[63:0] ← FS/GS_Segment_Base_Register;
Else
    DEST[31:0] ← FS/GS_Segment_Base_Register;
    DEST[63:32] ←0;
```

Flags Affected

None

C/C++ Compiler Intrinsic Equivalent

RDFSBASE unsigned int _readfsbase_u32(void);

RDFSBASE unsigned __int64 _readfsbase_u64(void);

RDGSBASE unsigned int _readgsbase_u32(void);

RDGSBASE unsigned __int64 _readgsbase_u64(void);

Protected Mode Exceptions

#UD	Always

Real-Address Mode Exceptions

#UD	Always

Virtual-8086 Mode Exceptions

#UD	Always

Compatibility Mode Exceptions

#UD	Always

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.FSGSBASE[bit 16] = 0. If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0

RDRAND—Read Random Number

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C7 /6 RDRAND r16	A	V/V	RDRND	Read a 16-bit random number and store in the destination register.
0F C7 /6 RDRAND r32	A	V/V	RDRND	Read a 32-bit random number and store in the destination register.
REX.W + 0F C7 /6 RDRAND r64	A	V/I	RDRND	Read a 64-bit random number and store in the destination register.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (w)	NA	NA	NA

Description

Loads a hardware generated random value and store it in the destination register. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random value has been returned, otherwise it is expected to loop and retry execution of RDRAND.

This instruction is available at all privilege levels. For virtualization supporting lock-step operation, a virtualization control exists that allows the virtual machine monitor to trap on the instruction. "RDRAND exiting" will be controlled by bit 11 of the secondary processor-based VM-execution control. A VMEXIT due to RDRAND will have exit reason 57 (decimal).

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF HW_RND_GEN.ready = 1
    THEN
```

```
        CASE of
            osize is 64: DEST[63:0] ← HW_RND_GEN.data;
            osize is 32: DEST[31:0] ← HW_RND_GEN.data;
            osize is 16: DEST[15:0] ← HW_RND_GEN.data;
        ESAC
        CF ← 1;
    ELSE
        CASE of
            osize is 64: DEST[63:0] ← 0;
            osize is 32: DEST[31:0] ← 0;
            osize is 16: DEST[15:0] ← 0;
        ESAC
        CF ← 0;
FI
OF, SF, ZF, AF, PF ← 0;
```

Flags Affected

All flags are affected

C/C++ Compiler Intrinsic Equivalent

RDRAND unsigned short _rdrand_u16(void);
RDRAND unsigned int _rdrand_u32(void);
RDRAND unsigned __int64 _rdrand_u64(void);

Protected Mode Exceptions

#UD	If the LOCK prefix is used. If the F2H or F3H prefix is used. If CPUID.01H:ECX.RDRND[bit 30] = 0

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

If the F2H or F3H prefix is used.
If CPUID.01H:ECX.RDRND[bit 30] = 0

Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.
If the F2H or F3H prefix is used.
If CPUID.01H:ECX.RDRND[bit 30] = 0

Compatibility Mode Exceptions

#UD If the LOCK prefix is used.
If the F2H or F3H prefix is used.
If CPUID.01H:ECX.RDRND[bit 30] = 0

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
If the F2H or F3H prefix is used.
If CPUID.01H:ECX.RDRND[bit 30] = 0

WRFSBASE/WRGSBASE—Write FS/GS Segment Base Register

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F AE /2 WRFSBASE r32	A	V/I	FSGSBASE	Write the 32-bit value in the source register to FS base register.
REX.W + F3 0F AE /2 WRFSBASE r64	A	V/I	FSGSBASE	Write the 64-bit value in the source register to FS base register.
F3 0F AE /3 WRGS-BASE *r32*	A	V/I	FSGSBASE	Write the 32-bit value in the source register to GS base register.
REX.W + F3 0F AE /3 WRGSBASE *r64*	A	V/I	FSGSBASE	Write the 64-bit value in the source register to GS base register.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (r)	NA	NA	NA

Description

Loads the source operand into the FS or GS segment base register. The source operand is a 32 or 64-bit general purpose register. The REX.W prefix indicates the operand size is 64 bit. If no REX.W prefix is used then the operand size is 32 bit and the upper 32 bits of the FS or GS base register will be written to 0.

This instruction is supported only in 64-bit sub mode of the IA-32e mode.

Operation

```
If osize = 64 then
    FS/GS_Segment_Base_Register ← SRC[63:0];
Else
    FS/GS_Segment_Base_Register[63:32] ← 0;
    FS/GS_Segment_Base_Register[31:0] ← SRC[31:0];
```

Flags Affected

None

C/C++ Compiler Intrinsic Equivalent

WRFSBASE void _writefsbase_u32(unsigned int);
WRFSBASE _writefsbase_u64(unsigned __int64);
WRGSBASE void _writegsbase_u32(unsigned int);
WRGSBASE _writegsbase_u64(unsigned __int64);

Protected Mode Exceptions

#UD	Always

Real-Address Mode Exceptions

#UD	Always

Virtual-8086 Mode Exceptions

#UD	Always

Compatibility Mode Exceptions

#UD	Always

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.FSGSBASE[bit 16] = 0.
	If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0
#GP(0)	If the SRC contains a non-canonical address.

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

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1, xmm2/m128	A	V/V	F16C	Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point value in ymm1.
VEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1, xmm2/m64	A	V/V	F16C	Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point value in xmm1.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts four/eight packed half precision (16-bits) floating-point values in the low-order 64/128 bits of an XMM/YMM register or 64/128-bit memory location to four/eight packed single-precision floating-point values and writes the converted values into the destination XMM/YMM register.

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

128-bit version: The source operand is a XMM register or 64-bit memory location. The destination operand is a XMM register. The upper bits (255:128) of the corresponding destination YMM register are zeroed.

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

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

Note: VEX.vvvv is reserved (must be 1111b).

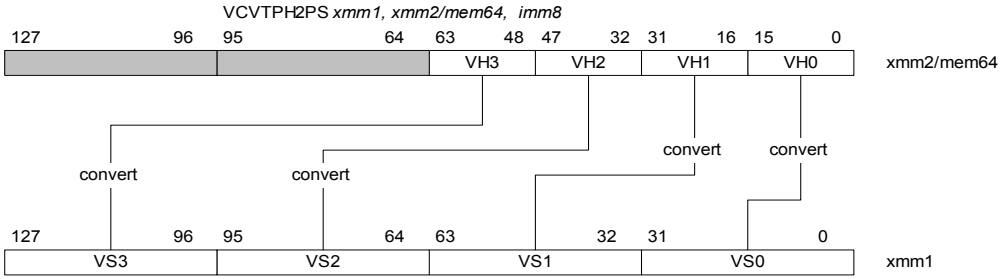


Figure 7-3. VCVTPH2PS (128-bit Version)

Operation

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

VCVTPH2PS (VEX.256 encoded version)

```
DEST[31:0] ←vCvt_h2s(SRC1[15:0]);
DEST[63:32] ←vCvt_h2s(SRC1[31:16]);
DEST[95:64] ←vCvt_h2s(SRC1[47:32]);
DEST[127:96] ←vCvt_h2s(SRC1[63:48]);
DEST[159:128] ←vCvt_h2s(SRC1[79:64]);
DEST[191:160] ←vCvt_h2s(SRC1[95:80]);
DEST[223:192] ←vCvt_h2s(SRC1[111:96]);
DEST[255:224] ←vCvt_h2s(SRC1[127:112]);
```

VCVTPH2PS (VEX.128 encoded version)

```
DEST[31:0] ←vCvt_h2s(SRC1[15:0]);
DEST[63:32] ←vCvt_h2s(SRC1[31:16]);
DEST[95:64] ←vCvt_h2s(SRC1[47:32]);
DEST[127:96] ←vCvt_h2s(SRC1[63:48]);
DEST[255:128] ←0
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

__m128 _mm_cvtph_ps (__m128i m1);
__m256 _mm256_cvtph_ps (__m128i m1)

SIMD Floating-Point Exceptions

Invalid;

Other Exceptions

Exceptions Type 11 (do not report #AC),

#UD	If VEX.W=1

VCVTPS2PH – Convert Single-Precision FP value to 16-bit FP value

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m128, ymm2, imm8	A	V/V	F16C	Convert eight packed single-precision floating-point value in ymm2 to packed half-precision (16-bit) floating-point value in xmm1/mem. Imm8 provides rounding controls.
VEX.128.66.0F3A.W0.1D /r ib VCVTPS2PH xmm1/m64, xmm2, imm8	A	V/V	F16C	Convert four packed single-precision floating-point value in xmm2 to packed half-precision (16-bit) floating point value in xmm1/mem. Imm8 provides rounding controls.

Instruction Operand Encoding

	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Convert four or eight packed single-precision floating values in first source operand to four or eight packed half-precision (16-bit) floating point values. The rounding mode are specified using the immediate field (imm8).

Underflow results (i.e. tiny results) are converted to denormals. MXCSR.FTZ is ignored. If a source element is denormal relative to input format with DM masked and at least one of PM or UM unmasked; a SIMD exception will be raised with DE, UE and PE set.

128-bit version: The source operand is a XMM register. The destination operand is a XMM register or 64-bit memory location. If destination operand is a register then the upper bits (255:64) of corresponding YMM register are zeroed.

256-bit version: The source operand is a YMM register. The destination operand is a XMM register or 128-bit memory location. If the destination operand is a register, the upper bits (255:128) of the corresponding YMM register are zeroed.

Note: VEX.vvvv is reserved (must be 1111b).

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

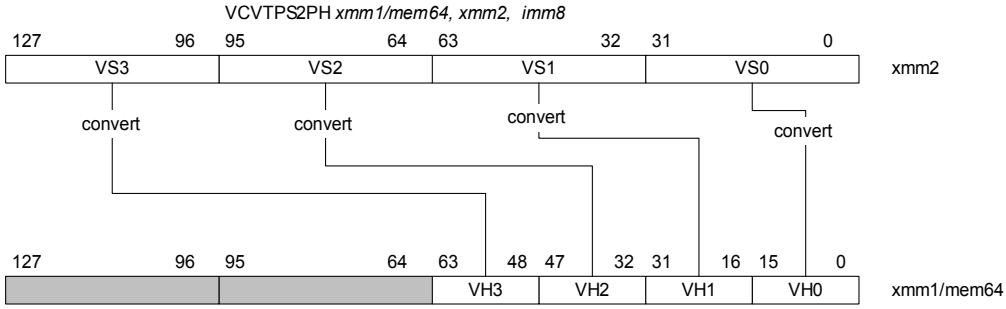


Figure 7-4. VCVTPS2PH (128-bit Version)

The immediate byte defines several bit fields that controls rounding operation. The effect and encoding of RC field are listed in Table 7-12.

Table 7-12. Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions

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

Operation

```
vCvt_s2h(SRC1[31:0])
{
IF Imm[2] = 0
THEN // using Imm[1:0] for rounding control, see Table 7-12
        RETURN Cvt_Single_Precision_To_Half_Precision_FP_Imm(SRC1[31:0]);
ELSE // using MXCSR.RC for rounding control
        RETURN Cvt_Single_Precision_To_Half_Precision_FP_Mxcsr(SRC1[31:0]);
FI;
}
```

VCVTPS2PH (VEX.256 encoded version)

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

VCVTPS2PH (VEX.128 encoded version)

```
DEST[15:0] ← vCvt_s2h(SRC1[31:0]);
DEST[31:16] ← vCvt_s2h(SRC1[63:32]);
DEST[47:32] ← vCvt_s2h(SRC1[95:64]);
DEST[63:48] ← vCvt_s2h(SRC1[127:96]);
DEST[255:64] ←0
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

```
__m128i _mm_cvtps_ph ( __m128 m1, const int imm);
__m128i _mm256_cvtps_ph(__m256 m1, const int imm);
```

SIMD Floating-Point Exceptions

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

Other Exceptions

Exceptions Type 11 (do not report #AC)

#UD	If VEX.W=1

This page was intentionally left blank.

APPENDIX A
INSTRUCTION SUMMARY

Most SSE/SSE2/SSE3/SSSE3/SSE4 Instructions have been promoted to support VEX.128 encodings which, for non-memory-store versions implies support for zeroing upper bits of YMM registers. Table A-1 summarizes the promotion status for existing instructions. The column "VEX.256" indicates whether 256-bit vector form of the instruction using the VEX.256 prefix encoding is supported. The column "VEX.128" indicates whether the instruction using VEX.128 prefix encoding is supported.

Table A-1. Promoted SSE/SSE2/SSE3/SSSE3/SSE4 Instructions

VEX.256 Encoding	VEX.128 Encoding	group	Instruction	If No, Reason?
yes	yes	YY 0F 1X	MOVUPS	
no	yes		MOVSS	scalar
yes	yes		MOVUPD	
no	yes		MOVSD	scalar
no	yes		MOVLPS	Note 1
no	yes		MOVLPD	Note 1
no	yes		MOVLHPS	Redundant with VPER-MILPS
yes	yes		MOVDDUP	
yes	yes		MOVSLDUP	
yes	yes		UNPCKLPS	
yes	yes		UNPCKLPD	
yes	yes		UNPCKHPS	
yes	yes		UNPCKHPD	
no	yes		MOVHPS	Note 1
no	yes		MOVHPD	Note 1
no	yes		MOVHLPS	Redundant with VPER-MILPS
yes	yes		MOVAPS	
yes	yes		MOVSHDUP	
yes	yes		MOVAPD	
no	no		CVTPI2PS	MMX
no	yes		CVTSI2SS	scalar
no	no		CVTPI2PD	MMX
no	yes		CVTSI2SD	scalar
no	yes		MOVNTPS	
no	yes		MOVNTPD	
no	no		CVTTPS2PI	MMX
no	yes		CVTTSS2SI	scalar
no	no		CVTTPD2PI	MMX
no	yes		CVTTSD2SI	scalar
no	no		CVTPS2PI	MMX
no	yes		CVTSS2SI	scalar
no	no		CVTPD2PI	MMX
no	yes		CVTSD2SI	scalar

VEX.256 Encoding	VEX.128 Encoding	group	Instruction	If No, Reason?
no	yes		UCOMISS	scalar
no	yes		UCOMISD	scalar
no	yes		COMISS	scalar
no	yes		COMISD	scalar
yes	yes	YY 0F 5X	MOVMSKPS	
yes	yes		MOVMSKPD	
yes	yes		SQRTPS	
no	yes		SQRTSS	scalar
yes	yes		SQRTPD	
no	yes		SQRTSD	scalar
yes	yes		RSQRTPS	
no	yes		RSQRTSS	scalar
yes	yes		RCPPS	
no	yes		RCPSS	scalar
yes	yes		ANDPS	
yes	yes		ANDPD	
yes	yes		ANDNPS	
yes	yes		ANDNPD	
yes	yes		ORPS	
yes	yes		ORPD	
yes	yes		XORPS	
yes	yes		XORPD	
yes	yes		ADDPS	
no	yes		ADDSS	scalar
yes	yes		ADDPD	
no	yes		ADDSD	scalar
yes	yes		MULPS	
no	yes		MULSS	scalar
yes	yes		MULPD	
no	yes		MULSD	scalar
yes	yes		CVTPS2PD	
no	yes		CVTSS2SD	scalar
yes	yes		CVTPD2PS	
no	yes		CVTSD2SS	scalar
yes	yes		CVTDQ2PS	

VEX.256 Encoding	VEX.128 Encoding	group	Instruction	If No, Reason?
yes	yes		CVTPS2DQ	
yes	yes		CVTTPS2DQ	
yes	yes		SUBPS	
no	yes		SUBSS	scalar
yes	yes		SUBPD	
no	yes		SUBSD	scalar
yes	yes		MINPS	
no	yes		MINSS	scalar
yes	yes		MINPD	
no	yes		MINSD	scalar
yes	yes		DIVPS	
no	yes		DIVSS	scalar
yes	yes		DIVPD	
no	yes		DIVSD	scalar
yes	yes		MAXPS	
no	yes		MAXSS	scalar
yes	yes		MAXPD	
no	yes		MAXSD	scalar
no	yes	YY 0F 6X	PUNPCKLBW	VI
no	yes		PUNPCKLWD	VI
no	yes		PUNPCKLDQ	VI
no	yes		PACKSSWB	VI
no	yes		PCMPGTB	VI
no	yes		PCMPGTW	VI
no	yes		PCMPGTD	VI
no	yes		PACKUSWB	VI
no	yes		PUNPCKHBW	VI
no	yes		PUNPCKHWD	VI
no	yes		PUNPCKHDQ	VI
no	yes		PACKSSDW	VI
no	yes		PUNPCKLQDQ	VI
no	yes		PUNPCKHQDQ	VI
no	yes		MOVD	scalar
no	yes		MOVQ	scalar
yes	yes		MOVDQA	

VEX.256 Encoding	VEX.128 Encoding	group	Instruction	If No, Reason?
yes	yes		MOVDQU	
no	yes	YY 0F 7X	PSHUFD	VI
no	yes		PSHUFHW	VI
no	yes		PSHUFLW	VI
no	yes		PCMPEQB	VI
no	yes		PCMPEQW	VI
no	yes		PCMPEQD	VI
yes	yes		HADDPD	
yes	yes		HADDPS	
yes	yes		HSUBPD	
yes	yes		HSUBPS	
no	yes		MOVD	VI
no	yes		MOVQ	VI
yes	yes		MOVDQA	
yes	yes		MOVDQU	
no	yes	YY 0F AX	LDMXCSR	
no	yes		STMXCSR	
yes	yes	YY 0F CX	CMPPS	
no	yes		CMPSS	scalar
yes	yes		CMPPD	
no	yes		CMPSD	scalar
no	yes		PINSRW	VI
no	yes		PEXTRW	VI
yes	yes		SHUFPS	
yes	yes		SHUFPD	
yes	yes	YY 0F DX	ADDSUBPD	
yes	yes		ADDSUBPS	
no	yes		PSRLW	VI
no	yes		PSRLD	VI
no	yes		PSRLQ	VI
no	yes		PADDQ	VI
no	yes		PMULLW	VI
no	no		MOVQ2DQ	MMX
no	no		MOVDQ2Q	MMX
no	yes		PMOVMSKB	VI

VEX.256 Encoding	VEX.128 Encoding	group	Instruction	If No, Reason?
no	yes		PSUBUSB	VI
no	yes		PSUBUSW	VI
no	yes		PMINUB	VI
no	yes		PAND	VI
no	yes		PADDUSB	VI
no	yes		PADDUSW	VI
no	yes		PMAXUB	VI
no	yes		PANDN	VI
no	yes	YY 0F EX	PAVGB	VI
no	yes		PSRAW	VI
no	yes		PSRAD	VI
no	yes		PAVGW	VI
no	yes		PMULHUW	VI
no	yes		PMULHW	VI
yes	yes		CVTPD2DQ	
yes	yes		CVTTPD2DQ	
yes	yes		CVTDQ2PD	
no	yes		MOVNTDQ	VI
no	yes		PSUBSB	VI
no	yes		PSUBSW	VI
no	yes		PMINSW	VI
no	yes		POR	VI
no	yes		PADDSB	VI
no	yes		PADDSW	VI
no	yes		PMAXSW	VI
no	yes		PXOR	VI
yes	yes	YY 0F FX	LDDQU	VI
no	yes		PSLLW	VI
no	yes		PSLLD	VI
no	yes		PSLLQ	VI
no	yes		PMULUDQ	VI
no	yes		PMADDWD	VI
no	yes		PSADBW	VI
no	yes		MASKMOVDQU	
no	yes		PSUBB	VI

VEX.256 Encoding	VEX.128 Encoding	group	Instruction	If No, Reason?
no	yes		PSUBW	VI
no	yes		PSUBD	VI
no	yes		PSUBQ	VI
no	yes		PADDB	VI
no	yes		PADDW	VI
no	yes		PADDD	VI
no	yes	SSSE3	PHADDW	VI
no	yes		PHADDSW	VI
no	yes		PHADDD	VI
no	yes		PHSUBW	VI
no	yes		PHSUBSW	VI
no	yes		PHSUBD	VI
no	yes		PMADDUBSW	VI
no	yes		PALIGNR	VI
no	yes		PSHUFB	VI
no	yes		PMULHRSW	VI
no	yes		PSIGNB	VI
no	yes		PSIGNW	VI
no	yes		PSIGND	VI
no	yes		PABSB	VI
no	yes		PABSW	VI
no	yes		PABSD	VI
yes	yes	SSE4.1	BLENDPS	
yes	yes		BLENDPD	
yes	yes		BLENDVPS	Note 2
yes	yes		BLENDVPD	Note 2
no	yes		DPPD	
yes	yes		DPPS	
no	yes		EXTRACTPS	Note 3
no	yes		INSERTPS	Note 3
no	yes		MOVNTDQA	
no	yes		MPSADBW	VI
no	yes		PACKUSDW	VI
no	yes		PBLENDVB	VI
no	yes		PBLENDW	VI

VEX.256 Encoding	VEX.128 Encoding	group	Instruction	If No, Reason?
no	yes		PCMPEQQ	VI
no	yes		PEXTRD	VI
no	yes		PEXTRQ	VI
no	yes		PEXTRB	VI
no	yes		PEXTRW	VI
no	yes		PHMINPOSUW	VI
no	yes		PINSRB	VI
no	yes		PINSRD	VI
no	yes		PINSRQ	VI
no	yes		PMAXSB	VI
no	yes		PMAXSD	VI
no	yes		PMAXUD	VI
no	yes		PMAXUW	VI
no	yes		PMINSB	VI
no	yes		PMINSD	VI
no	yes		PMINUD	VI
no	yes		PMINUW	VI
no	yes		PMOVSXxx	VI
no	yes		PMOVZXxx	VI
no	yes		PMULDQ	VI
no	yes		PMULLD	VI
yes	yes		PTEST	
yes	yes		ROUNDPD	
yes	yes		ROUNDPS	
no	yes		ROUNDSD	scalar
no	yes		ROUNDSS	scalar
no	yes	SSE4.2	PCMPGTQ	VI
no	no	SSE4.2	CRC32c	integer
no	yes		PCMPESTRI	VI
no	yes		PCMPESTRM	VI
no	yes		PCMPISTRI	VI
no	yes		PCMPISTRM	VI
no	no	SSE4.2	POPCNT	integer

Description of Column “If No, Reason?”

MMX: Instructions referencing MMX registers do not support VEX

Scalar: Scalar instructions are not promoted to 256-bit

integer: integer instructions are not promoted.

VI: "Vector Integer" instructions are not promoted to 256-bit

Note 1: MOVLPD/PS and MOVHPD/PS are not promoted to 256-bit. The equivalent functionality are provided by VINSERTF128 and VEXTRACTF128 instructions as the existing instructions have no natural 256b extension

Note 2: BLENDVPD and BLENDVPS are superseded by the more flexible VBLENDVPD and VBLENDVPS.

Note 3: It is expected that using 128-bit INSERTPS followed by a VINSERTF128 would be better than promoting INSERTPS to 256-bit (for example).

Table A-2. AVX, FMA and AES New Instructions

Opcode	Instruction	Description
66 0F 38 DE /r	AESDEC xmm1, xmm2/m128	Perform 1 round of AES decryption of xmm1 using the 128-bit round key from the xmm2/m128.
66 0F 38 DF /r	AESDECLAST xmm1, xmm2/m128	Perform the last round of AES decryption of xmm1 using the 128 bit round key from xmm2/m128.
VEX.NDS.128.66. 0F38 DE /r	VAESDEC xmm1, xmm2, xmm3/m128	Perform 1 round of AES decryption of xmm2 using the 128-bit round key from the xmm3/m128, and stores the result in xmm1.
VEX.NDS.128.66. 0F38 DF /r	VAESDECLAST xmm1, xmm2, xmm3/m128	Perform the last round of AES decryption of xmm2 using the 128 bit round key from xmm3/m128, and stores the result in xmm1.
66 0F 38 DC /r	AESENC xmm1, xmm2/m128	Perform 1 round of AES encryption of xmm1 using the 128-bit round key from the xmm2/m128.
66 0F 38 DD /r	AESENCLAST xmm1, xmm2/m128	Perform the last round of AES encryption of xmm1 using the 128 bit round key from xmm2/m128.
VEX.NDS.128.66. 0F38 DC /r	VAESENC xmm1, xmm2, xmm3/m128	Perform 1 round of AES encryption of xmm2 using the 128-bit round key from the xmm3/m128, and stores the result in xmm1.
VEX.NDS.128.66. 0F38 DD /r	VAESENCLAST xmm1, xmm2, xmm3/m128	Perform the last round of AES encryption of xmm2 using the 128 bit round key from xmm3/m128, and stores the result in xmm1.
66 0F 38 DB /r	AESIMC xmm1, xmm2/m128	Perform the InvMixColumn operation using xmm2/mem and store result in xmm1.
VEX.128.66.0F38 DB /r	VAESIMC xmm1, xmm2/m128	Perform the InvMixColumn operation using xmm2/mem and store result in xmm1.
66 0F 3A DF /r ib	AESKEYGENASSIST xmm1, xmm2/m128, imm8	Assist in AES round key generation using an immediate round control byte, a key specified in xmm2/m128 and stores the result in xmm1.
VEX.128.66.0F3A DF /r ib	VAESKEYGENASSIST xmm1, xmm2/m128, imm8	Assist in AES round key generation using an immediate round control byte, a key specified in xmm2/m128 and stores the result in xmm1.
VEX.256.66.0F38 1A /r	VBROADCASTF128 ymm1, m128	Broadcast 128-bit floating-point values in mem to low and high 128-bits in ymm1.
VEX.256.66.0F38 19/r	VBROADCASTSD ymm1, m64	Broadcast double-precision floating-point element in mem to four locations in ymm1.

Opcode	Instruction	Description
VEX.256.66.0F38 18 /r	VBROADCASTSS ymm1, m32	Broadcast single-precision floating-point element in mem to eight locations in ymm1.
VEX.128.66.0F38 18/r	VBROADCASTSS xmm1, m32	Broadcast single-precision floating-point element in mem to four locations in xmm1.
VEX.256.66.0F3A 19 /r ib	VEXTRACTF128 xmm1/m128, ymm2, imm8	Extracts 128-bits of packed floating-point values from ymm2 and store results in xmm1/mem.
VEX.DDS.128.66. 0F38.W1 98 /r	VFMADD132PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.128.66. 0F38.W1 A8 /r	VFMADD213PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm0, add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66. 0F38.W1 B8 /r	VFMADD231PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.
VEX.DDS.256.66. 0F38.W1 98 /r	VFMADD132PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, add to ymm1 and put result in ymm0.
VEX.DDS.256.66. 0F38.W1 A8 /r	VFMADD213PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm0, add to ymm2/mem and put result in ymm0.
VEX.DDS.256.66. 0F38.W1 B8 /r	VFMADD231PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, add to ymm0 and put result in ymm0.
VEX.DDS.128.66. 0F38.W0 98 /r	VFMADD132PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.128.66. 0F38.W0 A8 /r	VFMADD213PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm0, add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66. 0F38.W0 B8 /r	VFMADD231PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.
VEX.DDS.256.66. 0F38.W0 98 /r	VFMADD132PS ymm0, ymm1, ymm2/m256, ymm3	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, add to ymm1 and put result in ymm0.
VEX.DDS.256.66. 0F38.W0 A8 /r	VFMADD213PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm0, add to ymm2/mem and put result in ymm0.

Opcode	Instruction	Description
VEX.DDS.256.66. 0F38.W0 B8 /r	VFMADD231PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, add to ymm0 and put result in ymm0.
VEX.DDS.128.66. 0F38.W1 99 /r	VFMADD132SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.128.66. 0F38.W1 A9 /r	VFMADD213SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm0, add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66. 0F38.W1 B9 /r	VFMADD231SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.
VEX.DDS.128.66. 0F38.W0 99 /r	VFMADD132SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.128.66. 0F38.W0 A9 /r	VFMADD213SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm0, add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66. 0F38.W0 B9 /r	VFMADD231SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.
VEX.DDS.128.66. 0F38.W1 96 /r	VFMADDSUB132PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, add/subtract elements in xmm1 and put result in xmm0.
VEX.DDS.128.66. 0F38.W1 A6 /r	VFMADDSUB213PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm0, add/subtract elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66. 0F38.W1 B6 /r	VFMADDSUB231PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add/subtract elements in xmm0 and put result in xmm0.
VEX.DDS.256.66. 0F38.W1 96 /r	VFMADDSUB132PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, add/subtract elements in ymm1 and put result in ymm0.
VEX.DDS.256.66. 0F38.W1 A6 /r	VFMADDSUB213PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm0, add/subtract elements in ymm2/mem and put result in ymm0.

Opcode	Instruction	Description
VEX.DDS.256.66.0F38.W1 B6 /r	VFMADDSUB231PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm0 and ymm1, add/subtract elements in ymm2/mem and put result in ymm0.
VEX.DDS.128.66.0F38.W0 96 /r	VFMADDSUB132PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, add/subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 A6 /r	VFMADDSUB213PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm0, add/subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 B6 /r	VFMADDSUB231PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, add/subtract xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 96 /r	VFMADDSUB132PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, add/subtract ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 A6 /r	VFMADDSUB213PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm0, add/subtract ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W0 B6 /r	VFMADDSUB231PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, add/subtract ymm0 and put result in ymm0.
VEX.DDS.128.66.0F38.W1 97 /r	VFMSUBADD132PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, subtract/add elements in xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 A7 /r	VFMSUBADD213PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm0, subtract/add elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 B7 /r	VFMSUBADD231PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, subtract/add elements in xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W1 97 /r	VFMSUBADD132PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, subtract/add elements in ymm1 and put result in ymm0.

Opcode	Instruction	Description
VEX.DDS.256.66. 0F38.W1 A7 /r	VFMSUBADD213PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm0, subtract/add elements in ymm2/mem and put result in ymm0.
VEX.DDS.256.66. 0F38.W1 B7 /r	VFMSUBADD231PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, subtract/add elements in ymm0 and put result in ymm0.
VEX.DDS.128.66. 0F38.W0 97 /r	VFMSUBADD132PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, subtract/add xmm1 and put result in xmm0.
VEX.DDS.128.66. 0F38.W0 A7 /r	VFMSUBADD213PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm0, subtract/add xmm2/mem and put result in xmm0.
VEX.DDS.128.66. 0F38.W0 B7 /r	VFMSUBADD231PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, subtract/add xmm0 and put result in xmm0.
VEX.DDS.256.66. 0F38.W0 97 /r	VFMSUBADD132PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, subtract/add ymm1 and put result in ymm0.
VEX.DDS.256.66. 0F38.W0 A7 /r	VFMSUBADD213PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm0, subtract/add ymm2/mem and put result in ymm0.
VEX.DDS.256.66. 0F38.W0 B7 /r	VFMSUBADD231PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, subtract/add ymm0 and put result in ymm0.
VEX.DDS.128.66. 0F38.W1 9A /r	VFMSUB132PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66. 0F38.W1 AA /r	VFMSUB213PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm0, subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66. 0F38.W1 BA /r	VFMSUB231PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.
VEX.DDS.256.66. 0F38.W1 9A /r	VFMSUB132PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, subtract ymm1 and put result in ymm0.

Opcode	Instruction	Description
VEX.DDS.256.66.0F38.W1 AA /r	VFMSUB213PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm0, subtract ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W1 BA /r	VFMSUB231PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, subtract ymm0 and put result in ymm0.
VEX.DDS.128.66.0F38.W0 9A /r	VFMSUB132PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 AA /r	VFMSUB213PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm0, subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 BA /r	VFMSUB231PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 9A /r	VFMSUB132PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, subtract ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 AA /r	VFMSUB213PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm0, subtract ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W0 BA /r	VFMSUB231PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, subtract ymm0 and put result in ymm0.
VEX.DDS.128.66.0F38.W1 9B /r	VFMSUB132SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 AB /r	VFMSUB213SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm0, subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 BB /r	VFMSUB231SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 9B /r	VFMSUB132SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 AB /r	VFMSUB213SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm0, subtract xmm2/mem and put result in xmm0.

Opcode	Instruction	Description
VEX.DDS.128.66. 0F38.W0 BB /r	VFMSUB231SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.
VEX.DDS.128.66. 0F38.W1 9C /r	VFNMADD132PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and add to xmm1. Put the result in xmm0.
VEX.DDS.128.66. 0F38.W1 AC /r	VFNMADD213PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm0, negate the multiplication result and add to xmm2/mem. Put the result in xmm0.
VEX.DDS.128.66. 0F38.W1 BC /r	VFNMADD231PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and add to xmm0. Put the result in xmm0.
VEX.DDS.256.66. 0F38.W1 9C /r	VFNMADD132PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and add to ymm1. Put the result in ymm0.
VEX.DDS.256.66. 0F38.W1 AC /r	VFNMADD213PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm0, negate the multiplication result and add to ymm2/mem. Put the result in ymm0.
VEX.DDS.256.66. 0F38.W1 BC /r	VFNMADD231PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and add to ymm0. Put the result in ymm0.
VEX.DDS.128.66. 0F38.W0 9C /r	VFNMADD132PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and add to xmm1. Put the result in xmm0.
VEX.DDS.128.66. 0F38.W0 AC /r	VFNMADD213PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm0, negate the multiplication result and add to xmm2/mem. Put the result in xmm0.
VEX.DDS.128.66. 0F38.W0 BC /r	VFNMADD231PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and add to xmm0. Put the result in xmm0.
VEX.DDS.256.66. 0F38.W0 9C /r	VFNMADD132PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and add to ymm1. Put the result in ymm0.

Opcode	Instruction	Description
VEX.DDS.256.66.0F38.W0 AC /r	VFNMADD213PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm0, negate the multiplication result and add to ymm2/mem. Put the result in ymm0.
VEX.DDS.256.66.0F38.W0 BC /r	VFNMADD231PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and add to ymm0. Put the result in ymm0.
VEX.DDS.128.66.0F38.W1 9D /r	VFNMADD132SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm0 and xmm2/mem, negate the multiplication result and add to xmm1. Put the result in xmm0.
VEX.DDS.128.66.0F38.W1 AD /r	VFNMADD213SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm0, negate the multiplication result and add to xmm2/mem. Put the result in xmm0.
VEX.DDS.128.66.0F38.W1 BD /r	VFNMADD231SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and add to xmm0. Put the result in xmm0.
VEX.DDS.128.66.0F38.W0 9D /r	VFNMADD132SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm0 and xmm2/mem, negate the multiplication result and add to xmm1. Put the result in xmm0.
VEX.DDS.128.66.0F38.W0 AD /r	VFNMADD213SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm0, negate the multiplication result and add to xmm2/mem. Put the result in xmm0.
VEX.DDS.128.66.0F38.W0 BD /r	VFNMADD231SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and add to xmm0. Put the result in xmm0.
VEX.DDS.128.66.0F38.W1 9E /r	VFNMSUB132PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1. Put the result in xmm0.
VEX.DDS.128.66.0F38.W1 AE /r	VFNMSUB213PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm0, negate the multiplication result and subtract xmm2/mem. Put the result in xmm0.

Opcode	**Instruction**	**Description**
VEX.DDS.128.66.0F38.W1 BE /r	VFNMSUB231PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0. Put the result in xmm0.
VEX.DDS.256.66.0F38.W1 9E /r	VFNMSUB132PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and subtract ymm1. Put the result in ymm0.
VEX.DDS.256.66.0F38.W1 AE /r	VFNMSUB213PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm0, negate the multiplication result and subtract ymm2/mem. Put the result in ymm0.
VEX.DDS.256.66.0F38.W1 BE /r	VFNMSUB231PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and subtract ymm0. Put the result in ymm0.
VEX.DDS.128.66.0F38.W0 9E /r	VFNMSUB132PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1. Put the result in xmm0.
VEX.DDS.128.66.0F38.W0 AE /r	VFNMSUB213PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm0, negate the multiplication result and subtract xmm2/mem. Put the result in xmm0.
VEX.DDS.128.66.0F38.W0 BE /r	VFNMSUB231PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0. Put the result in xmm0.
VEX.DDS.256.66.0F38.W0 9E /r	VFNMSUB132PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and subtract ymm1. Put the result in ymm0.
VEX.DDS.256.66.0F38.W0 AE /r	VFNMSUB213PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm0, negate the multiplication result and subtract ymm2/mem. Put the result in ymm0.
VEX.DDS.256.66.0F38.W0 BE /r	VFNMSUB231PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and subtract ymm0. Put the result in ymm0.

Opcode	Instruction	Description
VEX.DDS.128.66. 0F38.W1 9F /r	VFNMSUB132SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1. Put the result in xmm0.
VEX.DDS.128.66. 0F38.W1 AF /r	VFNMSUB213SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm0, negate the multiplication result and subtract xmm2/mem. Put the result in xmm0.
VEX.DDS.128.66. 0F38.W1 BF /r	VFNMSUB231SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0. Put the result in xmm0.
VEX.DDS.128.66. 0F38.W0 9F /r	VFNMSUB132SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1. Put the result in xmm0.
VEX.DDS.128.66. 0F38.W0 AF /r	VFNMSUB213SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm0, negate the multiplication result and subtract xmm2/mem. Put the result in xmm0.
VEX.DDS.128.66. 0F38.W0 BF /r	VFNMSUB231SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0. Put the result in xmm0.
VEX.NDS.256.66. 0F3A 18 /r ib	VINSERTF128 ymm1, ymm2, xmm3/m128, imm8	Insert 128-bits of packed floating-point values from xmm3/mem and the remaining values from ymm2 into ymm1
VEX.NDS.128.66. 0F38 2C /r	VMASKMOVPS xmm1, xmm2, m128	Load packed single-precision values from mem using mask in xmm2 and store in xmm1
VEX.NDS.256.66. 0F38 2C /r	VMASKMOVPS ymm1, ymm2, m256	Load packed single-precision values from mem using mask in ymm2 and store in ymm1
VEX.NDS.128.66. 0F38 2D/r	VMASKMOVPD xmm1, xmm2, m128	Load packed double-precision values from mem using mask in xmm2 and store in xmm1
VEX.NDS.256.66. 0F38 2D /r	VMASKMOVPD ymm1, ymm2, m256	Load packed double-precision values from mem using mask in ymm2 and store in ymm1
VEX.NDS.128.66. 0F38 2E /r	VMASKMOVPS m128, xmm1, xmm2	Store packed single-precision values from xmm2 using mask in xmm1

Opcode	Instruction	Description
VEX.NDS.256.66.0F38 2E /r	VMASKMOVPS m256, ymm1, ymm2	Store packed single-precision values from ymm2 mask in ymm1
VEX.NDS.128.66.0F38 2F /r	VMASKMOVPD m128, xmm1, xmm2	Store packed double-precision values from xmm2 using mask in xmm1
VEX.NDS.256.66.0F38 2F /r	VMASKMOVPD m256, ymm1, ymm2	Store packed double-precision values from ymm2 using mask in ymm1
VEX.NDS.128.66.0F38 0D /r	VPERMILPD xmm1, xmm2, XMM3/m128	Permute Double-Precision Floating-Point values in xmm2 using controls from xmm3/mem and store result in xmm1
VEX.128.66.0F3A 05 /r ib	VPERMILPD xmm1, xmm2/m128, imm8	Permute Double-Precision Floating-Point values in xmm2/mem using controls from imm8 and store result in xmm1
VEX.NDS.256.66.0F38 0D /r	VPERMILPD ymm1, ymm2, ymm3/m256	Permute Double-Precision Floating-Point values in ymm2 using controls from xmm3/mem and store result in ymm1
VEX.256.66.0F3A 05 /r ib	VPERMILPD ymm1, ymm2/m256 imm8	Permute Double-Precision Floating-Point values in ymm2/mem using controls from imm8 and store result in ymm1
VEX.NDS.128.66.0F38 0C /r	VPERMILPS xmm1, xmm2, xmm3/m128	Permute Single-Precision Floating-Point values in xmm2 using controls from xmm3/mem and store result in xmm1
VEX.128.66.0F3A 04 /r ib	VPERMILPS xmm1, xmm2/m128, imm8	Permute Single-Precision Floating-Point values in xmm2/mem using controls from imm8 and store result in xmm1
VEX.NDS.256.66.0F38 0C /r	VPERMILPS ymm1, ymm2, YMM/m256	Permute Single-Precision Floating-Point values in ymm2 using controls from ymm3/mem and store result in ymm1
VEX.256.66.0F3A 04 /r ib	VPERMILPS ymm1, ymm2/m256, imm8	Permute Single-Precision Floating-Point values in ymm2/mem using controls from imm8 and store result in ymm1
VEX.NDS.256.66.0F3A 06 /r ib	VPERM2F128 ymm1, ymm2, ymm3/m256, imm8	Permute 128-bit floating-point fields in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1
66 0F 3A 44 /r ib	PCLMULQDQ xmm1, xmm2/m128, imm8	Carry-less multiplication of a pair of quad-word selected by an immediate byte from xmm2/m128 and xmm1, stores the 128-bit result in xmm1.
VEX.256.66.0F38 0E /r	VTESTPS ymm1, ymm2/m256	Set ZF if ymm2/mem AND ymm1 result is all 0s in packed single-precision sign bits. Set CF if ymm2/mem AND NOT ymm1 result is all 0s in packed single-precision sign bits.

Opcode	Instruction	Description
VEX.256.66.0F38 0F /r	VTESTPD ymm1, ymm2/m256	Set ZF if ymm2/mem AND ymm1 result is all 0s in packed double-precision sign bits. Set CF if ymm2/mem AND NOT ymm1 result is all 0s in packed double-precision sign bits.
VEX.128.66.0F38 0E /r	VTESTPS xmm1, xmm2/m128	Set ZF if xmm2/mem AND xmm1 result is all 0s in packed single-precision sign bits. Set CF if xmm2/mem AND NOT xmm1 result is all 0s in packed single-precision sign bits.
VEX.128.66.0F38 0F /r	VTESTPD xmm1, xmm2/m128	Set ZF if xmm2/mem AND xmm1 result is all 0s in packed single precision sign bits. Set CF if xmm2/mem AND NOT xmm1 result is all 0s in packed double-precision sign bits.
VEX.256.0F 77	VZEROALL	Zero all YMM registers
VEX.128.0F 77	VZEROUPPER	Zero upper 128 bits of all YMM registers

Table A-3. Other New Instructions

Opcode	Instruction	Description
F3 0F AE /0	RDFSBASE r32	Read FS base register and place the 32-bit result in the destination register.
REX.W + F3 0F AE /0	RDFSBASE r64	Read FS base register and place the 64-bit result in the destination register.
F3 0F AE /1	RDGSBASE r32	Read GS base register and place the 32-bit result in destination register.
REX.W + F3 0F AE /1	RDGSBASE r64	Read GS base register and place the 64-bit result in destination register.
0F C7 /6	RDRAND r16	Read a 16-bit random number and store in the destination register.
0F C7 /6	RDRAND r32	Read a 32-bit random number and store in the destination register.
REX.W + 0F C7 /6	RDRAND r64	Read a 64-bit random number and store in the destination register.
F3 0F AE /2	WRFSBASE r32	Write the 32-bit value in the source register to FS base register.
REX.W + F3 0F AE /2	WRFSBASE r64	Write the 64-bit value in the source register to FS base register.
F3 0F AE /3	WRGSBASE r32	Write the 32-bit value in the source register to GS base register.

Opcode	Instruction	Description
REX.W + F3 0F AE /3	WRGSBASE r64	Write the 64-bit value in the source register to GS base register.
VEX.256.66.0F38. W0 13 /r	VCVTPH2PS ymm1, xmm2/m128	Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point value in ymm1.
VEX.128.66.0F38. W0 13 /r	VCVTPH2PS xmm1, xmm2/m64	Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point value in xmm1.
VEX.256.66.0F3A. W0 1D /r ib	VCVTPS2PH xmm1/m128, ymm2, imm8	Convert eight packed single-precision floating-point value in ymm2 to packed half-precision (16-bit) floating-point value in xmm1/mem. Imm8 provides rounding controls.
VEX.128.66.0F3A. W0.1D /r ib	VCVTPS2PH xmm1/m64, xmm2, imm8	Convert four packed single-precision floating-point value in xmm2 to packed half-precision (16-bit) floating point value in xmm1/mem. Imm8 provides rounding controls.

APPENDIX B
INSTRUCTION OPCODE MAP

Use the opcode tables in this chapter to interpret IA-32 and intel 64 architecture object code. 1-byte, 2-byte and 3-byte opcode encodings are used to encode integer, system, MMX technology, SSE/SSE2/SSE3/SSSE3/SSE4, and VMX instructions. Maps for these instructions are given in Table B-2 through Table B-6.

NOTE

All blanks in opcode maps are reserved and must not be used. Do not depend on the operation of undefined or blank opcodes.

B.1 USING OPCODE TABLES

Tables in this appendix list opcodes of instructions (including required instruction prefixes, opcode extensions in associated ModR/M byte). Blank cells in the tables indicate opcodes that are reserved or undefined.

The opcode map tables are organized by hex values of the upper and lower 4 bits of an opcode byte. For 1-byte encodings (Table B-2), use the four high-order bits of an opcode to index a row of the opcode table; use the four low-order bits to index a column of the table. For 2-byte opcodes beginning with 0FH (Table B-3), skip any instruction prefixes, the 0FH byte (0FH may be preceded by 66H, F2H, or F3H) and use the upper and lower 4-bit values of the next opcode byte to index table rows and columns. Similarly, for 3-byte opcodes beginning with 0F38H or 0F3AH (Table B-4), skip any instruction prefixes, 0F38H or 0F3AH and use the upper and lower 4-bit values of the third opcode byte to index table rows and columns. See Section B.2.4, “Opcode Look-up Examples for One, Two, and Three-Byte Opcodes.”

When a ModR/M byte provides opcode extensions, this information qualifies opcode execution. For information on how an opcode extension in the ModR/M byte modifies the opcode map in Table B-2 and Table B-3, see Section B.4.

B.2 KEY TO ABBREVIATIONS

Operands are identified by a two-character code of the form Zz. The first character, an uppercase letter, specifies the addressing method; the second character, a lower-case letter, specifies the type of operand.

B.2.1 Codes for Addressing Method

The following abbreviations are used to document addressing methods:

A Direct address: the instruction has no ModR/M byte; the address of the operand is encoded in the instruction. No base register, index register, or scaling factor can be applied (for example, far JMP (EA)).

C The reg field of the ModR/M byte selects a control register (for example, MOV (0F20, 0F22)).

D The reg field of the ModR/M byte selects a debug register (for example, MOV (0F21,0F23)).

E A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

F EFLAGS/RFLAGS Register.

G The reg field of the ModR/M byte selects a general register (for example, AX (000)).

H The VEX.vvvv field of the VEX prefix selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. For legacy SSE encodings this operand does not exist, changing the instruction to destructive form.

I Immediate data: the operand value is encoded in subsequent bytes of the instruction.

J The instruction contains a relative offset to be added to the instruction pointer register (for example, JMP (0E9), LOOP).

L The upper 4 bits of the 8-bit immediate selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. (the MSB is ignored in 32-bit mode)

M The ModR/M byte may refer only to memory (for example, BOUND, LES, LDS, LSS, LFS, LGS, CMPXCHG8B).

N The R/M field of the ModR/M byte selects a packed-quadword, MMX technology register.

O The instruction has no ModR/M byte. The offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied (for example, MOV (A0–A3)).

P The reg field of the ModR/M byte selects a packed quadword MMX technology register.

Q A ModR/M byte follows the opcode and specifies the operand. The operand is either an MMX technology register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.

R The R/M field of the ModR/M byte may refer only to a general register (for example, MOV (0F20-0F23)).

S The reg field of the ModR/M byte selects a segment register (for example, MOV (8C,8E)).

U The R/M field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.

V The reg field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.

W A ModR/M byte follows the opcode and specifies the operand. The operand is either a 128-bit XMM register, a 256-bit YMM register (determined by operand type), or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.

X Memory addressed by the DS:rSI register pair (for example, MOVS, CMPS, OUTS, or LODS).

Y Memory addressed by the ES:rDI register pair (for example, MOVS, CMPS, INS, STOS, or SCAS).

B.2.2 Codes for Operand Type

The following abbreviations are used to document operand types:

a Two one-word operands in memory or two double-word operands in memory, depending on operand-size attribute (used only by the BOUND instruction).

b Byte, regardless of operand-size attribute.

c Byte or word, depending on operand-size attribute.

d Doubleword, regardless of operand-size attribute.

dq Double-quadword, regardless of operand-size attribute.

p 32-bit, 48-bit, or 80-bit pointer, depending on operand-size attribute.

pd 128-bit or 256-bit packed double-precision floating-point data.

pi Quadword MMX technology register (for example: mm0).

ps 128-bit or 256-bit packed single-precision floating-point data.

q Quadword, regardless of operand-size attribute.

qq Quad-Quadword (256-bits), regardless of operand-size attribute.

s 6-byte or 10-byte pseudo-descriptor.

sd Scalar element of a 128-bit double-precision floating data.

ss Scalar element of a 128-bit single-precision floating data.

si Doubleword integer register (for example: eax).

v Word, doubleword or quadword (in 64-bit mode), depending on operand-size attribute.

w Word, regardless of operand-size attribute.

x dq or qq based on the operand-size attribute.

y Doubleword or quadword (in 64-bit mode), depending on operand-size attribute.

z Word for 16-bit operand-size or doubleword for 32 or 64-bit operand-size.

B.2.3 Register Codes

When an opcode requires a specific register as an operand, the register is identified by name (for example, AX, CL, or ESI). The name indicates whether the register is 64, 32, 16, or 8 bits wide.

A register identifier of the form eXX or rXX is used when register width depends on the operand-size attribute. eXX is used when 16 or 32-bit sizes are possible; rXX is used when 16, 32, or 64-bit sizes are possible. For example: eAX indicates that the AX register is used when the operand-size attribute is 16 and the EAX register is used when the operand-size attribute is 32. rAX can indicate AX, EAX or RAX.

When the REX.B bit is used to modify the register specified in the reg field of the opcode, this fact is indicated by adding "/x" to the register name to indicate the additional possibility. For example, rCX/r9 is used to indicate that the register could either be rCX or r9. Note that the size of r9 in this case is determined by the operand size attribute (just as for rCX).

B.2.4 Opcode Look-up Examples for One, Two, and Three-Byte Opcodes

This section provides examples that demonstrate how opcode maps are used.

B.2.4.1 One-Byte Opcode Instructions

The opcode map for 1-byte opcodes is shown in Table B-2. The opcode map for 1-byte opcodes is arranged by row (the least-significant 4 bits of the hexadecimal value) and column (the most-significant 4 bits of the hexadecimal value). Each entry in the table lists one of the following types of opcodes:

- Instruction mnemonics and operand types using the notations listed in Section B.2
- Opcodes used as an instruction prefix

For each entry in the opcode map that corresponds to an instruction, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section B.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. Operand types are listed according to notations listed in Section B.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table B-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction prefix or entries for instructions without operands that use ModR/M (for example: 60H, PUSHA; 06H, PUSH ES).

Example B-1. Look-up Example for 1-Byte Opcodes

Opcode 030500000000H for an ADD instruction is interpreted using the 1-byte opcode map (Table B-2) as follows:

- The first digit (0) of the opcode indicates the table row and the second digit (3) indicates the table column. This locates an opcode for ADD with two operands.
- The first operand (type Gv) indicates a general register that is a word or doubleword depending on the operand-size attribute. The second operand (type Ev) indicates a ModR/M byte follows that specifies whether the operand is a word or doubleword general-purpose register or a memory address.
- The ModR/M byte for this instruction is 05H, indicating that a 32-bit displacement follows (00000000H). The reg/opcode portion of the ModR/M byte (bits 3-5) is 000, indicating the EAX register.

The instruction for this opcode is ADD EAX, mem_op, and the offset of mem_op is 00000000H.

Some 1- and 2-byte opcodes point to group numbers (shaded entries in the opcode map table). Group numbers indicate that the instruction uses the reg/opcode bits in the ModR/M byte as an opcode extension (refer to Section B.4).

B.2.4.2 Two-Byte Opcode Instructions

The two-byte opcode map shown in Table B-3 includes primary opcodes that are either two bytes or three bytes in length. Primary opcodes that are 2 bytes in length begin with an escape opcode 0FH. The upper and lower four bits of the second opcode byte are used to index a particular row and column in Table B-3.

Two-byte opcodes that are 3 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and the escape opcode (0FH). The upper and lower four bits of the third

byte are used to index a particular row and column in Table B-3 (except when the second opcode byte is the 3-byte escape opcodes 38H or 3AH; in this situation refer to Section B.2.4.3).

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section B.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. The operand types are listed according to notations listed in Section B.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table B-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction without operands that are encoded using ModR/M (for example: 0F77H, EMMS).

Example B-2. Look-up Example for 2-Byte Opcodes

Look-up opcode 0FA4050000000003H for a SHLD instruction using Table B-3.

- The opcode is located in row A, column 4. The location indicates a SHLD instruction with operands Ev, Gv, and Ib. Interpret the operands as follows:
  - Ev: The ModR/M byte follows the opcode to specify a word or doubleword operand.
  - Gv: The reg field of the ModR/M byte selects a general-purpose register.
  - Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The third byte is the ModR/M byte (05H). The mod and opcode/reg fields of ModR/M indicate that a 32-bit displacement is used to locate the first operand in memory and eAX as the second operand.
- The next part of the opcode is the 32-bit displacement for the destination memory operand (00000000H). The last byte stores immediate byte that provides the count of the shift (03H).
- By this breakdown, it has been shown that this opcode represents the instruction: SHLD DS:00000000H, EAX, 3.

B.2.4.3 Three-Byte Opcode Instructions

The three-byte opcode maps shown in Table B-4 and Table B-5 includes primary opcodes that are either 3 or 4 bytes in length. Primary opcodes that are 3 bytes in length begin with two escape bytes 0F38H or 0F3A. The upper and lower four bits of the third opcode byte are used to index a particular row and column in Table B-4 or Table B-5.

Three-byte opcodes that are 4 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and two escape bytes (0F38H or 0F3AH). The upper and lower four bits

of the fourth byte are used to index a particular row and column in Table B-4 or Table B-5.

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into the following case:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in B.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. The operand types are listed according to notations listed in Section B.2.

Example B-3. Look-up Example for 3-Byte Opcodes

Look-up opcode 660F3A0FC108H for a PALIGNR instruction using Table B-5.

- 66H is a prefix and 0F3AH indicate to use Table B-5. The opcode is located in row 0, column F indicating a PALIGNR instruction with operands Vdq, Wdq, and Ib. Interpret the operands as follows:
  - Vdq: The reg field of the ModR/M byte selects a 128-bit XMM register.
  - Wdq: The R/M field of the ModR/M byte selects either a 128-bit XMM register or memory location.
  - Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The next byte is the ModR/M byte (C1H). The reg field indicates that the first operand is XMM0. The mod shows that the R/M field specifies a register and the R/M indicates that the second operand is XMM1.
- The last byte is the immediate byte (08H).
- By this breakdown, it has been shown that this opcode represents the instruction: PALIGNR XMM0, XMM1, 8.

B.2.4.4 VEX Prefix Instructions

Instructions that include a VEX prefix are organized relative to the 2-byte and 3-byte opcode maps, based on the VEX.mmmmm field encoding of implied 0F, 0F38H, 0F3AH, respectively. Each entry in the opcode map of a VEX-encoded instruction is based on the value of the opcode byte, similar to non-VEX-encoded instructions.

A VEX prefix includes several bit fields that encode implied 66H, F2H, F3H prefix functionality (VEX.pp) and operand size/opcode information (VEX.L). See chapter 4 for details.

Opcode tables A2-A6 include both instructions with a VEX prefix and instructions without a VEX prefix. Many entries are only made once, but represent both the VEX and non-VEX forms of the instruction. If the VEX prefix is present all the operands are valid and the mnemonic is usually prefixed with a "v". If the VEX prefix is not present the VEX.vvvv operand is not available.

A few instructions exist only in VEX form and these are marked with a superscript "v".

Operand size of VEX prefix instructions can be determined by the operand type code. 128-bit vectors are indicated by 'dq', 256-bit vectors are indicated by 'qq', and instructions with operands supporting either 128 or 256-bit, determined by VEX.L, are indicated by 'x'. For example, the entry "VMOVUPD Vx,Wx" indicates both VEX.L=0 and VEX.L=1 are supported.

B.2.5 Superscripts Utilized in Opcode Tables

Table B-1 contains notes on particular encodings. These notes are indicated in the following opcode maps by superscripts. Gray cells indicate instruction groupings.

Table B-1. Superscripts Utilized in Opcode Tables

Superscript Symbol	Meaning of Symbol
1A	Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (refer to Section B.4, "Opcode Extensions For One-Byte And Two-byte Opcodes").
1B	Use the 0F0B opcode (UD2 instruction) or the 0FB9H opcode when deliberately trying to generate an invalid opcode exception (#UD).
1C	Some instructions added in the Pentium III processor may use the same two-byte opcode. If the instruction has variations, or the opcode represents different instructions, the ModR/M byte will be used to differentiate the instruction. For the value of the ModR/M byte needed to decode the instruction, see Table B-6. These instructions include SFENCE, STMXCSR, LDMXCSR, FXRSTOR, and FXSAVE, as well as PREFETCH and its variations.
i64	The instruction is invalid or not encodable in 64-bit mode. 40 through 4F (single-byte INC and DEC) are REX prefix combinations when in 64-bit mode (use FE/FF Grp 4 and 5 for INC and DEC).
o64	Instruction is only available when in 64-bit mode.
d64	When in 64-bit mode, instruction defaults to 64-bit operand size and cannot encode 32-bit operand size.
f64	The operand size is forced to a 64-bit operand size when in 64-bit mode (prefixes that change operand size are ignored for this instruction in 64-bit mode).
v	VEX form only exists. There is no legacy SSE form of the instruction.
v1	VEX128 & SSE forms only exist (no VEX256), when can't be inferred from the data size.

B.3 ONE, TWO, AND THREE-BYTE OPCODE MAPS

See Table B-2 through Table B-5 below. The tables are multiple page presentations. Rows and columns with sequential relationships are placed on facing pages to make look-up tasks easier. Note that table footnotes are not presented on each page. Table footnotes for each table are presented on the last page of the table.

Table B-2. One-byte Opcode Map: (00H — F7H) *

	0	1	2	3	4	5	6	7
0	ADD Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH ES^{i64}	POP ES^{i64}
1	ADC Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH SS^{i64}	POP SS^{i64}
2	AND Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=ES (Prefix)	DAA^{i64}
3	XOR Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=SS (Prefix)	AAA^{i64}
4	INC^{i64} general register / REX^{o64} Prefixes eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	PUSH^{d64} general register rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA^{i64}/ PUSHAD^{i64}	POPA^{i64}/ POPAD^{i64}	BOUND^{i64} Gv, Ma	ARPL^{i64} Ew, Gw MOVSXD^{o64} Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc^{f64}, Jb - Short-displacement jump on condition O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1^{1A} Eb, Ib	Ev, Iz	Eb, Ib^{i64}	Ev, Ib	TEST Eb, Gb	Ev, Gv	XCHG Eb, Gb	Ev, Gv
9	NOP PAUSE(F3) XCHG r8, rAX	XCHG word, double-word or quad-word register with rAX rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
A	MOV AL, Ob	rAX, Ov	Ob, AL	Ov, rAX	MOVS/B Xb, Yb	MOVS/W/D/Q Xv, Yv	CMPS/B Xb, Yb	CMPS/W/D Xv, Yv
B	MOV immediate byte into byte register AL/R8L, Ib	CL/R9L, Ib	DL/R10L, Ib	BL/R11L, Ib	AH/R12L, Ib	CH/R13L, Ib	DH/R14L, Ib	BH/R15L, Ib
C	Shift Grp 2^{1A} Eb, Ib	Ev, Ib	RETN^{f64} Iw	RETN^{f64}	LES^{i64} Gz, Mp VEX+2byte	LDS^{i64} Gz, Mp VEX+1byte	Grp 11^{1A} - MOV Eb, Ib	Ev, Iz
D	Shift Grp 2^{1A} Eb, 1	Ev, 1	Eb, CL	Ev, CL	AAM^{i64} Ib	AAD^{i64} Ib		XLAT/ XLATB
E	LOOPNE^{f64}/ LOOPNZ^{f64} Jb	LOOPE^{f64}/ LOOPZ^{f64} Jb	LOOP^{f64} Jb	JrCXZ^{f64}/ Jb	IN AL, Ib	eAX, Ib	OUT Ib, AL	Ib, eAX
F	LOCK (Prefix)		REPNE (Prefix)	REP/REPE (Prefix)	HLT	CMC	Unary Grp 3^{1A} Eb	Ev

Table B-2. One-byte Opcode Map: (08H — FFH) *

	8	9	A	B	C	D	E	F
0	OR Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH CS[i64]	2-byte escape (Table A-3)
1	SBB Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH DS[i64]	POP DS[i64]
2	SUB Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=CS (Prefix)	DAS[i64]
3	CMP Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=DS (Prefix)	AAS[i64]
4	DEC[i64] general register / REX[o64] Prefixes eAX REX.W	eCX REX.WB	eDX REX.WX	eBX REX.WXB	eSP REX.WR	eBP REX.WRB	eSI REX.WRX	eDI REX.WRXB
5	POP[d64] into general register rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSH[d64] Iz	IMUL Gv, Ev, Iz	PUSH[d64] Ib	IMUL Gv, Ev, Ib	INS/ INSB Yb, DX	INS/ INSW/ INSD Yz, DX	OUTS/ OUTSB DX, Xb	OUTS/ OUTSW/ OUTSD DX, Xz
7	Jcc[f64], Jb- Short displacement jump on condition S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
8	MOV Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	MOV Ev, Sw	LEA Gv, M	MOV Sw, Ew	Grp 1A[1A] POP[d64] Ev
9	CBW/ CWDE/ CDQE	CWD/ CDQ/ CQO	CALLF[i64] Ap	FWAIT/ WAIT	PUSHF/D/Q[d64]/ Fv	POPF/D/Q[d64]/ Fv	SAHF	LAHF
A	TEST AL, Ib	rAX, Iz	STOS/B Yb, AL	STOS/W/D/Q Yv, rAX	LODS/B AL, Xb	LODS/W/D/Q rAX, Xv	SCAS/B AL, Yb	SCAS/W/D/Q rAX, Xv
B	MOV immediate word or double into word, double, or quad register rAX/r8, Iv	rCX/r9, Iv	rDX/r10, Iv	rBX/r11, Iv	rSP/r12, Iv	rBP/r13, Iv	rSI/r14, Iv	rDI/r15 , Iv
C	ENTER Iw, Ib	LEAVE[d64]	RETF Iw	RETF	INT 3	INT Ib	INTO[i64]	IRET/D/Q
D	ESC (Escape to coprocessor instruction set)							
E	CALL[f64] Jz	JMP near[f64] Jz	far[i64] AP	short[f64] Jb	IN AL, DX	eAX, DX	OUT DX, AL	DX, eAX
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4[1A]	INC/DEC Grp 5[1A]

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table B-3. Two-byte Opcode Map: 00H — 77H (First Byte is 0FH) *

	pfx	0	1	2	3	4	5	6	7
0		Grp 6[1A]	Grp 7[1A]	LAR Gv, Ew	LSL Gv, Ew		SYSCALL[o64]	CLTS	SYSRET[o64]
1		vmovups	vmovups	vmovlps Vq, Hq, Mq vmovhlps Vq, Hq, Uq	vmovlps Mq, Vq	vunpcklps Vps, Wq Vx, Hx, Wx	vunpckhps Vps, Wq Vx, Hx, Wx	vmovhps[v1] Vdq, Hq, Mq vmovlhps Vdq, Hq, Uq	vmovhps[v1] Mq, Vq
	66	vmovupd	vmovupd Wpd,Vpd	vmovlpd Vq, Hq, Mq	vmovlpd Mq, Vq	vunpcklpd Vx,Hx,Wx	vunpckhpd Vx,Hx,Wx	vmovhpd[v1] Vdq, Hq, Mq	vmovhpd[v1] Mq, Vq
	F3	vmovss Vss, Wss Vss, Hss, Uss	vmovss Wss, Vss Uss, Hss, Vss	vmovsldup Vx, Wx				vmovshdup Vx, Wx	
	F2	vmovsd Vsd, Wsd Usd, Hsd, Vsd	vmovsd Vsd, Wsd Usd, Hsd, Vsd	vmovddup Vx, Wx					
2	2	MOV Rd, Cd	MOV Rd, Dd	MOV Cd, Rd	MOV Dd, Rd				
3	3	WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER	SYSEXIT		GETSEC
4	4	CMOVcc, (Gv, Ev) - Conditional Move							
		O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
5		vmovmskps Gy, U	vsqrtps	vrsqrtps	vrcpps	vandps Vps, Hps, Wps	vandnps Vps, Hps, Wps	vorps Vps, Hps, Wps	vxorps Vps, Hps, Wps
	66	vmovmskpd Gy,U	vsqrtpd Wpd,Vpd			vandpd Wpd, Hpd, Vpd	vandnpd Wpd, Hpd, Vpd	vorpd Wpd, Hpd, Vpd	vxorpd Wpd, Hpd, Vpd
	F3		vsqrtss Vss, Hss, Wss	vrsqrtss Vss, Hss, Wss	vrcpss Vss, Hss, Wss				
	F2		vsqrtsd Vsd, Hsd, Wsd						
6		punpcklbw Pq, Qd	punpcklwd Pq, Qd	punpckldq Pq, Qd	packsswb Pq, Qq	pcmpgtb Pq, Qq	pcmpgtw Pq, Qq	pcmpgtd Pq, Qq	packuswb Pq, Qq
	66	vpunpcklbw Vdq, Hdq, Wdq	vpunpcklwd Vdq, Hdq, Wdq	vpunpckldq Vdq, Hdq, Wdq	vpacksswb Vdq, Hdq,Wdq	vpcmpgtb Vdq, Hdq, Wdq	vpcmpgtw Vdq, Hdq, Wdq	vpcmpgtd Vdq, Hdq, Wdq	vpackuswb Vdq, Hdq, Wdq
	F3								
7		pshufw Pq, Qq, Ib	(Grp 12[1A])	(Grp 13[1A])	(Grp 14[1A])	pcmpeqb Pq, Qq	pcmpeqw Pq, Qq	pcmpeqd Pq, Qq	emms vzeroupper[v] vzeroall[v]
	66	vpshufd Vdq,Hdq,Wdq,Ib				vpcmpeqb Vdq, Hdq, Wdq	vpcmpeqw Vdq, Hdq, Wdq	vpcmpeqd Vdq, Hdq, Wdq	
	F3	vpshufhw Vdq,Hdq,Wdq,Ib							
	F2	vpshuflw Vdq,Hdq,Wdq,Ib							

Table B-3. Two-byte Opcode Map: 08H — 7FH (First Byte is 0FH) *

	pfx	8	9	A	B	C	D	E	F
0		INVD	WBINVD		2-byte Illegal Opcodes UD2[1B]		NOP Ev		
1		Prefetch[1C] (Grp 16[1A])							NOP Ev
2		vmovaps Vps, Wps	vmovaps Wps, Vps	cvtpi2ps Vps, Qpi	vmovntps Mps, Vps	cvttps2pi Ppi, Wps	cvtps2pi Ppi, Wps	vucomiss Vss, Wss	vcomiss Vss, Wss
	66	vmovapd Vpd, Wpd	vmovapd Wpd,Vpd	cvtpi2pd Vpd, Qpi	vmovntpd Mpd, Vpd	cvttpd2pi Ppi, Wpd	cvtpd2pi Qpi, Wpd	vucomisd Vsd, Wsd	vcomisd Vsd, Wsd
	F3			vcvtsi2ss Vss, Hss, Ey		vcvttss2si Gy, Wss	vcvtss2si Gy, Wss		
	F2			vcvtsi2sd Vsd, Hsd, Ey		vcvttsd2si Gy, Wsd	vcvtsd2si Gy, Wsd		
3	3	3-byte escape (Table A-4)		3-byte escape (Table A-5)					
4	4	CMOVcc(Gv, Ev) - Conditional Move							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
5		vaddps Vps, Hps, Wps	vmulps Vps, Hps, Wps	vcvtps2pd	vcvtdq2ps	vsubps Vps, Hps, Wps	vminps Vps, Hps, Wps	vdivps Vps, Hps, Wps	vmaxps Vps, Hps, Wps
	66	vaddpd Vpd, Hpd, Wpd	vmulpd Vpd, Hpd, Wpd	vcvtpd2ps Vps, Wpd	vcvtps2dq Vdq, Wps	vsubpd Vpd, Hpd, Wpd	vminpd Vpd, Hpd, Wpd	vdivpd Vpd, Hpd, Wpd	vmaxpd Vpd, Hpd, Wpd
	F3	vaddss Vss, Hss, Wss	vmulss Vss, Hss, Wss	vcvtss2sd Vsd, Hx, Wss	vcvttps2dq Vdq, Wps	vsubss Vss, Hss, Wss	vminss Vss, Hss, Wss	vdivss Vss, Hss, Wss	vmaxss Vss, Hss, Wss
	F2	vaddsd Vsd, Hsd, Wsd	vmulsd Vsd, Hsd, Wsd	vcvtsd2ss Vss, Hx, Wsd		vsubsd Vsd, Hsd, Wsd	vminsd Vsd, Hsd, Wsd	vdivsd Vsd, Hsd, Wsd	vmaxsd Vsd, Hsd, Wsd
6		punpckhbw Pq, Qd	punpckhwd Pq, Qd	punpckhdq Pq, Qd	packssdw Pq, Qd			movd/q Pd, Ey	movq Pq, Qq
	66	vpunpckhbw Vdq, Hdq, Wdq	vpunpckhwd Vdq, Hdq, Wdq	vpunpckhdq Vdq, Hdq, Wdq	vpackssdw Vdq, Hdq, Wdq	vpunpcklqdq Vdq, Hdq, Wdq	vpunpckhqdq Vdq, Hdq, Wdq	vmovd/q Vy, Ey	vmovdqa Vx, Wx
	F3								vmovdqu Vx, Wx
7		VMREAD Ey, Gy	VMWRITE Gy, Ey					movd/q Ey, Pd	movq Qq, Pq
	66					vhaddpd Vpd, Hpd, Wpd	vhsubpd Vpd, Hpd, Wpd	vmovd/q Ey, Vy	vmovdqa Wx,Vx
	F3							vmovq Vq, Wq	vmovdqu Wx,Vx
	F2					vhaddps Vps, Hps, Wps	vhsubps Vps, Hps, Wps		

Table B-3. Two-byte Opcode Map: 80H — F7H (First Byte is 0FH) *

	pfx	0	1	2	3	4	5	6	7
8		Jcc[f64], Jz - Long-displacement jump on condition							
		O	NO	B/CNAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
9		SETcc, Eb - Byte Set on condition							
		O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
A		PUSH[d64] FS	POP[d64] FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
B		CMPXCHG Eb, Gb	CMPXCHG Ev, Gv	LSS Gv, Mp	BTR Ev, Gv	LFS Gv, Mp	LGS Gv, Mp	MOVZX Gv, Eb	MOVZX Gv, Ew
C		XADD Eb, Gb	XADD Ev, Gv	vcmpps Vps,Hps,Wps,Ib	movnti My, Gy	pinsrw Pq,Ry/Mw,Ib	pextrw Gd, Nq, Ib	vshufps Vps,Hps,Wps,Ib	Grp 9[1A]
	66			vcmppd Vpd,Hpd,Wpd,Ib		vpinsrw Vdq,Hdq,Ry/Mw,Ib	vpextrw Gd, Udq, Ib	vshufpd Vpd,Hpd,Wpd,Ib	
	F3			vcmpss Vss,Hss,Wss,Ib					
	F2			vcmpsd Vsd,Hsd,Wsd,Ib					
D			psrlw Pq, Qq	psrld Pq, Qq	psrlq Pq, Qq	paddq Pq, Qq	pmullw Pq, Qq		pmovmskb Gd, Nq
	66	vaddsubpd Vpd, Hpd, Wpd	vpsrlw Vdq, Hdq, Wdq	vpsrld Vdq, Hdq, Wdq	vpsrlq Vdq, Hdq, Wdq	vpaddq Vdq, Hdq, Wdq	vpmullw Vdq, Hdq, Wdq	vmovq Wq, Vq	vpmovmskb Gd, Udq
	F3							movq2dq Vdq, Nq	
	F2	vaddsubps Vps, Hps, Wps						movdq2q Pq, Uq	
E		pavgb Pq, Qq	psraw Pq, Qq	psrad Pq, Qq	pavgw Pq, Qq	pmulhuw Pq, Qq	pmulhw Pq, Qq		movntq Mq, Pq
	66	vpavgb Vdq, Hdq, Wdq	vpsraw Vdq, Hdq, Wdq	vpsrad Vdq, Hdq, Wdq	vpavgw Vdq, Hdq, Wdq	vpmulhuw Vdq, Hdq, Wdq	vpmulhw Vdq, Hdq, Wdq	vcvttpd2dq Vx, Wpd	vmovntdq Mx, Vx
	F3							vcvtdq2pd Vx, Wpd	
	F2							vcvtpd2dq Vx, Wpd	
F			psllw Pq, Qq	pslld Pq, Qq	psllq Pq, Qq	pmuludq Pq, Qq	pmaddwd Pq, Qq	psadbw Pq, Qq	maskmovq Pq, Nq
	66		vpsllw Vdq, Hdq, Wdq	vpslld Vdq, Hdq, Wdq	vpsllq Vdq, Hdq, Wdq	vpmuludq Vdq, Hdq, Wdq	vpmaddwd Vdq, Hdq, Wdq	vpsadbw Vdq, Hdq, Wdq	vmaskmovdqu Vdq, Udq
	F2	vlddqu Vx, Mx							

Table B-3. Two-byte Opcode Map: 88H — FFH (First Byte is 0FH) *

	pfx	8	9	A	B	C	D	E	F
8		Jcc[f64], Jz - Long-displacement jump on condition							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
9		SETcc, Eb - Byte Set on condition							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
A		PUSH[d64] GS	POP[d64] GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL	(Grp 15[1A])[1C]	IMUL Gv, Ev
B		JMPE (reserved for emulator on IPF)	Grp 10[1A] Invalid Opcode[1B]	Grp 8[1A] Ev, Ib	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSX Gv, Eb	MOVSX Gv, Ew
	F3	POPCNT Gv, Ev							
C		BSWAP							
		RAX/EAX/ R8/R8D	RCX/ECX/ R9/R9D	RDX/EDX/ R10/R10D	RBX/EBX/ R11/R11D	RSP/ESP/ R12/R12D	RBP/EBP/ R13/R13D	RSI/ESI/ R14/R14D	RDI/EDI/ R15/R15D
D		psubusb Pq, Qq	psubusw Pq, Qq	pminub Pq, Qq	pand Pq, Qq	paddusb Pq, Qq	paddusw Pq, Qq	pmaxub Pq, Qq	pandn Pq, Qq
	66	vpsubusb Vdq, Hdq, Wdq	vpsubusw Vdq, Hdq, Wdq	vpminub Vdq, Hdq, Wdq	vpand Vdq, Hdq, Wdq	vpaddusb Vdq, Hdq, Wdq	vpaddusw Vdq, Hdq, Wdq	vpmaxub Vdq, Hdq, Wdq	vpandn Vdq, Hdq, Wdq
	F3								
	F2								
E		psubsb Pq, Qq	psubsw Pq, Qq	pminsw Pq, Qq	por Pq, Qq	paddsb Pq, Qq	paddsw Pq, Qq	pmaxsw Pq, Qq	pxor Pq, Qq
	66	vpsubsb Vdq, Hdq, Wdq	vpsubsw Vdq, Hdq, Wdq	vpminsw Vdq, Hdq, Wdq	vpor Vdq, Hdq, Wdq	vpaddsb Vdq, Hdq, Wdq	vpaddsw Vdq, Hdq, Wdq	vpmaxsw Vdq, Hdq, Wdq	vpxor Vdq, Hdq, Wdq
	F3								
	F2								
F		psubb Pq, Qq	psubw Pq, Qq	psubd Pq, Qq	psubq Pq, Qq	paddb Pq, Qq	paddw Pq, Qq	paddd Pq, Qq	
	66	vpsubb Vdq, Hdq, Wdq	vpsubw Vdq, Hdq, Wdq	vpsubd Vdq, Hdq, Wdq	vpsubq Vdq, Hdq, Wdq	vpaddb Vdq, Hdq, Wdq	vpaddw Vdq, Hdq, Wdq	vpaddd Vdq, Hdq, Wdq	
	F2								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table B-4. Three-byte Opcode Map: 00H — F7H (First Two Bytes are 0F 38H) *

	pfx	0	1	2	3	4	5	6	7
0		pshufb Pq, Qq	phaddw Pq, Qq	phaddd Pq, Qq	phaddsw Pq, Qq	pmaddubsw Pq, Qq	phsubw Pq, Qq	phsubd Pq, Qq	phsubsw Pq, Qq
	66	vpshufb Vdq, Hdq, Wdq	vphaddw Vdq, Hdq, Wdq	vphaddd Vdq, Hdq, Wdq	vphaddsw Vdq, Hdq, Wdq	vpmaddubsw Vdq, Hdq, Wdq	vphsubw Vdq, Hdq, Wdq	vphsubd Vdq, Hdq, Wdq	vphsubsw Vdq, Hdq, Wdq
1	66	vpblendvb Vdq, Hdq, Wdq			vcvtph2ps[V] Vx, Wx, Ib	vblendvps Vx, Hx, Wx	vblendvpd Vx, Hx, Wx		vptest Vx, Wx
2	66	vpmovsxbw Vdq, Udq/Mq	vpmovsxbd Vdq, Udq/Md	vpmovsxbq Vdq, Udq/Mw	vpmovsxwd Vdq, Udq/Mq	vpmovsxwq Vdq, Udq/Md	vpmovsxdq Vdq, Udq/Mq		
3	66	vpmovzxbw Vdq, Udq/Mq	vpmovzxbd Vdq, Udq/Md	vpmovzxbq Vdq, Udq/Mw	vpmovzxwd Vdq, Udq/Mq	vpmovzxwq Vdq, Udq/Md	vpmovzxdq Vdq, Udq/Mq		vpcmpgtq Vdq, Hdq, Wdq
4	66	vpmulld Vdq, Hdq, Wdq	vphminposuw Vdq, Wdq						
5									
6									
7									
8	66	INVEPT Gy, Mdq	INVVPID Gy, Mdq						
9									
A									
B									
C									
D									
E									
F		MOVBE Gy, My	MOVBE My, Gy						
	66	MOVBE Gw, Mw	MOVBE Mw, Gw						
	F3								
	F2	CRC32 Gd, Eb	CRC32 Gd, Ey						
	66 & F2	CRC32 Gd, Eb	CRC32 Gd, Ew						

Table B-4. Three-byte Opcode Map: 08H — FFH (First Two Bytes are 0F 38H) *

	pfx	8	9	A	B	C	D	E	F
0		psignb Pq, Qq	psignw Pq, Qq	psignd Pq, Qq	pmulhrsw Pq, Qq				
	66	vpsignb Vdq, Hdq, Wdq	vpsignw Vdq, Hdq, Wdq	vpsignd Vdq, Hdq, Wdq	vpmulhrsw Vdq, Hdq, Wdq	vpermilps[v] Vx,Hx,Wx	vpermilpd[v] Vx,Hx,Wx	vtestps[v] Vx, Wx	vtestpd[v] Vx, Wx
1						pabsb Pq, Qq	pabsw Pq, Qq	pabsd Pq, Qq	
	66	vbroadcastss[v] Vx, Md	vbroadcastsd[v] Vqq, Mq	vbroadcastf128[v] Vqq, Mdq		vpabsb Vdq, Hdq, Wdq	vpabsw Vdq, Hdq, Wdq	vpabsd Vdq, Hdq, Wdq	
2	66	vpmuldq Vdq, Hdq, Wdq	vpcmpeqq Vdq, Hdq, Wdq	vmovntdqa Vdq, Hdq, Mdq	vpackusdw Vdq, Hdq, Wdq	vmaskmovps[v] Vx,Hx,Mx	vmaskmovpd[v] Vx,Hx,Mx	vmaskmovps[v] Mx,Vx,Hx	vmaskmovpd[v] Mx,Vx,Hx
3	66	vpminsb Vdq, Hdq, Wdq	vpminsd Vdq, Hdq, Wdq	vpminuw Vdq, Hdq, Wdq	vpminud Vdq, Hdq, Wdq	vpmaxsb Vdq, Hdq, Wdq	vpmaxsd Vdq, Hdq, Wdq	vpmaxuw Vdq, Hdq, Wdq	vpmaxud Vdq, Hdq, Wdq
4									
5									
6									
7									
8									
9									
A									
B									
C									
D	66				VAESIMC Vdq, Wdq	VAESENC Vdq,Hdq,Wdq	VAESENCLAST Vdq,Hdq,Wdq	VAESDEC Vdq,Hdq,Wdq	VAESDECLAST Vdq,Hdq,Wdq
E									
F									
	66								
	F3								
	F2								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table B-5. Three-byte Opcode Map: 00H — F7H (First two bytes are 0F 3AH) *

	pfx	0	1	2	3	4	5	6	7
0	66					vpermilps[V] Vx, Wx, Ib	vpermilpd[V] Vx, Wx, Ib	vperm2f128[V] Vqq,Hqq,Wqq,Ib	
1	66					vpextrb Rd/Mb, Vdq, Ib	vpextrw Rd/Mw, Vdq, Ib	vpextrd/q Ey, Vdq, Ib	vextractps Ed, Vdq, Ib
2	66	vpinsrb Vdq,Hdq, Ry/Mb,Ib	vinsertps Vdq,Hdq, Udq/Md,Ib	vpinsrd/q Vdq,Hdq,Ey,Ib					
3									
4	66	vdpps Vx,Hx,Wx,Ib	vdppd Vx,Hx,Wx,Ib	vmpsadbw Vdq,Hdq,Wdq,Ib		vpclmulqdq Vdq,Hdq,Wdq,Ib			
5									
6	66	vpcmpestrmdq, Wdq, Ib	vpcmpestri Vdq, Wdq, Ib	vpcmpistrm Vdq, Wdq, Ib	vpcmpistri Vdq, Wdq, Ib				
7									
8									
9									
A									
B									
C									
D									
E									
F									

Table B-5. Three-byte Opcode Map: 08H — FFH (First Two Bytes are 0F 3AH) *

	pfx	8	9	A	B	C	D	E	F
0									palignr Pq, Qq, Ib
	66	vroundps Vx,Hx,Wx,Ib	vroundpd Vx,Hx,Wx,Ib	vroundss Vss,Hss,Wss,Ib	vroundsd Vsd,Hsd,Wsd,Ib	vblendps Vx,Hx,Wx,Ib	vblendpd Vx,Hx,Wx,Ib	vpblendw Vdq,Hdq,Wdq,Ib	vpalignr Vdq,Hdq,Wdq,Ib
1	66	vinsertf128[V] Vqq,Hqq,Wqq,Ib	vextractf128[V] Wdq,Vqq,Ib				vcvtps2ph[V] Wx, Vx, Ib		
2									
3									
4	66			vblendvps[V] Vx,Hx,Wx,Lx	vblendvpd[V] Vx,Hx,Wx,Lx	vpblendvb[V] Vdq,Hdq,Wdq, Ldq			
5									
6									
7									
8									
9									
A									
B									
C									
D	66								VAESKEYGEN Vdq, Wdq, Ib
E									
F									

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

B.4 OPCODE EXTENSIONS FOR ONE-BYTE AND TWO-BYTE OPCODES

Some 1-byte and 2-byte opcodes use bits 3-5 of the ModR/M byte (the nnn field in Figure B-1) as an extension of the opcode.

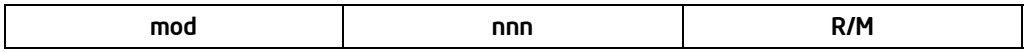


Figure B-1. ModR/M Byte nnn Field (Bits 5, 4, and 3)

Opcodes that have opcode extensions are indicated in Table B-6 and organized by group number. Group numbers (from 1 to 16, second column) provide a table entry point. The encoding for the r/m field for each instruction can be established using the third column of the table.

B.4.1 Opcode Look-up Examples Using Opcode Extensions

An Example is provided below.

Example B-3. Interpreting an ADD Instruction

An ADD instruction with a 1-byte opcode of 80H is a Group 1 instruction:

- Table B-6 indicates that the opcode extension field encoded in the ModR/M byte for this instruction is 000B.
- The r/m field can be encoded to access a register (11B) or a memory address using a specified addressing mode (for example: mem = 00B, 01B, 10B).

Example B-2. Looking Up 0F01C3H

Look up opcode 0F01C3 for a VMRESUME instruction by using Table B-2, Table B-3 and Table B-6:

- 0F tells us that this instruction is in the 2-byte opcode map.
- 01 (row 0, column 1 in Table B-3) reveals that this opcode is in Group 7 of Table B-6.
- C3 is the ModR/M byte. The first two bits of C3 are 11B. This tells us to look at the second of the Group 7 rows in Table B-6.
- The Op/Reg bits [5,4,3] are 000B. This tells us to look in the 000 column for Group 7.
- Finally, the R/M bits [2,1,0] are 011B. This identifies the opcode as the VMRESUME instruction.

B.4.2 Opcode Extension Tables

See Table B-6 below.

Table B-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number *

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
				000	001	010	011	100	101	110	111
80-83	1	mem, 11B		ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
8F	1A	mem, 11B		POP							
C0,C1 reg, imm D0, D1 reg, 1 D2, D3 reg, CL	2	mem, 11B		ROL	ROR	RCL	RCR	SHL/SAL	SHR		SAR
F6, F7	3	mem, 11B		TEST Ib/Iz		NOT	NEG	MUL AL/rAX	IMUL AL/rAX	DIV AL/rAX	IDIV AL/rAX
FE	4	mem, 11B		INC Eb	DEC Eb						
FF	5	mem, 11B		INC Ev	DEC Ev	CALLN[f64] Ev	CALLF Ep	JMPN[f64] Ev	JMPF Ep	PUSH[d64] Ev	
0F 00	6	mem, 11B		SLDT Rv/Mw	STR Rv/Mw	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
0F 01	7	mem		SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Mw/Rv		LMSW Ew	INVLPG Mb
		11B		VMCALL (001) VMLAUNCH (010) VMRESUME (011) VMXOFF (100)	MONITOR (000) MWAIT (001)	XGETBV (000) XSETBV (001)					SWAPGS[o64] (000) RDTSCP (001)
0F BA	8	mem, 11B						BT	BTS	BTR	BTC
0F C7	9	mem			CMPXCH8B Mq CMPXCHG16B Mdq					VMPTRLD Mq	VMPTRST Mq
			66							VMCLEAR Mq	
			F3							VMXON Mq	VMPTRST Mq
		11B								RDRAND Rv	
0F B9	10	mem									
		11B									
C6	11	mem, 11B		MOV Eb, Ib							
C7		mem		MOV Ev, Iz							
		11B									

Table B-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number *

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
				000	001	010	011	100	101	110	111
0F 71	12	mem									
		11B				psrlw Nq, Ib		psraw Nq, Ib		psllw Nq, Ib	
			66			vpsrlw Hdq,Udq,Ib		vpsraw Hdq,Udq,Ib		vpsllw Hdq,Udq,Ib	
0F 72	13	mem									
		11B				psrld Nq, Ib		psrad Nq, Ib		pslld Nq, Ib	
			66			vpsrld Hdq,Udq,Ib		vpsrad Hdq,Udq,Ib		vpslld Hdq,Udq,Ib	
0F 73	14	mem									
		11B				psrlq Nq, Ib				psllq Nq, Ib	
			66			vpsrlq Hdq,Udq,Ib	vpsrldq Hdq,Udq,Ib			vpsllq Hdq,Udq,Ib	vpslldq Hdq,Udq,Ib
0F AE	15	mem		fxsave	fxrstor	ldmxcsr	stmxcsr	XSAVE	XRSTOR	XSAVEOPT	clflush
		11B							lfence	mfence	sfence
			F3	RDFSBASE Ry	RDGSBASE Ry	WRFSBASE Ry	WRGSBASE Ry				
0F 18	16	mem		prefetch NTA	prefetch T0	prefetch T1	prefetch T2				
		11B									

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

INDEX

A

B

C

D

E

F

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z